Natural ISPF

Programmer's Guide

Version 8.2.6

November 2016

This document applies to Natural ISPF Version 8.2.6.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: ISP-PROG-826-20161124

Table of Contents

Preface

Natural ISPF is the application development tool of Software AG and provides a full range of application programming and system programming facilities in any mainframe environment that includes Natural.

This documentation describes the special programming facilities provided by Natural ISPF to help you realize advanced application development. For a description of the Natural ISPF user interface and object-types, consult the *Natural ISPF User's Guide*.

This documentation is organized under the following headings:

Macro Facility	Describes the Natural ISPF macro feature that allows you to use the Natural language to generate text of any kind.
Incore Database	Explains the Natural ISPF Incore Database facility which enables the Natural programmer to maintain complex data structures in the user memory and to perform complex actions on these structures.
Open NSPF	Provides information about the Open NSPF facility which enables you to modify and enhance Natural ISPF according to site-specific needs.
Application Programming Interface	Describes the programs which can be used in Natural applications (or in Open NSPF routines) to access Natural ISPF internal information.
Authorization Classes	Lists the available authorization classes and the Natural ISPF objects they refer to.

1 Macro Facility

Natural ISPF provides a macro feature that allows you to use the Natural language to generate text of any kind. In a process known as macro expansion, text is generated, which can consist of substituting variables, repeating blocks, generating blocks conditionally, even performing screen or file I/Os.

The macro feature is useful when you are creating different sources, all of the same structure but with different content. The macro feature thus supports you in editing programs and other sources, offering many uses within Natural ISPF.

This chapter describes macro syntax and the objects in which the macro feature can be used, including some examples.

Macro Syntax

The macro feature is an extension of the Natural language and consists of two types of statement, identified in the source by the macro character (in the given examples, the paragraph sign (§) is used). These statements are:

Processing Statements

Executed during macro expansion; these statements must be preceded by the macro character followed by a blank. The full Natural language is available for processing statements. You can work in either structured or report mode as normal.

Example

```
0010 § MOVE 'PERSONNEL' TO #FILE-NAME (A32)
0020 § MOVE 'NAME'       TO #KEY      (A32)
```

Text Lines

Copied to the generated output of the macro; text lines can contain variables that are substituted by their current values during macro expansion. Variables in text lines are identified by the macro character, for example:

```
0030 READ §#FILE-NAME BY §#KEY
0040 WRITE 'RECORD READ'
```

Variables can also be used as part of names. To concatenate a variable value with the rest of the name during macro expansion, you must use the vertical line as concatenation character (|).

Note: If your keyboard does not have the vertical line, use the character that has hexadecimal value 4F.

Examples

- For example, using the values in the examples above, the line:

```
0030 FIND §#FILE-NAME|-VIEW BY §#KEY
```

  generates the following text:

```
0030 FIND PERSONNEL-VIEW BY NAME
```

  Note: The concatenation character need only be typed after the macro variable, and need not be typed before a variable to be substituted after a concatenated string.

- For example, the JCL line:

```
0040 //   DD DSN=§#DSNIN|(§#MEMBER|)
```

  generates the following text using current variable values:

```
0040 //   DD DSN=ISP.COMN.DATA(ISPJCL)
```

  In this example, the concatenation character is used to concatenate the parentheses.

  If the macro character itself is to appear in the macro output, you must write a double macro character in the source.

- For example, the following line:

```
0010 * this macro uses the §§ char as macro char.
```

  generates the following text:

```
0010 * this macro uses the § char as macro char.
```

Examples of Macro Usage

1. Straight Substitution of Variables

The source lines:

```
§ MOVE 'PERSONNEL' TO #FILE-NAME (A32)
§ MOVE 'NAME'      TO #KEY       (A32)
READ §#FILE-NAME BY §#KEY
WRITE 'RECORD READ' §#KEY
```

produce the following output text:

```
READ PERSONNEL BY NAME
WRITE 'RECORD READ' NAME
```

2. Define Loops

The following source lines generate a Natural data definition:

```
§ DEFINE DATA LOCAL
§ 1 #FIELD    (A32/5) INIT <'MAKE','MODEL','COLOR'>
§ 1 #I        (N2)
§ END-DEFINE
§ *
DEFINE DATA LOCAL
1 AUTOMOBILES-VIEW
§ FOR #I = 1 TO 5
§ IF #FIELD(#I) = ' ' ESCAPE BOTTOM END-IF
2 §#FIELD(#I)
§ END-FOR
END-DEFINE
```

The following text lines are produced:

```
DEFINE DATA LOCAL
1 AUTOMOBILES-VIEW
2 MAKE
2 MODEL
2 COLOR
END-DEFINE
```

3. Screen I/O

The following source lines prompt you for a member name:

```
§ RESET #MEMBER(A8)
§ INPUT #MEMBER
//§*INIT-USER|A JOB
//ASM EXEC ASMM,MEM=§#MEMBER
```

The resulting output text using input member name `EDRBPM` is:

```
//MBEA JOB
//ASM EXEC ASMM,MEM=EDRBPM
```

4. File I/O

The following source lines read the first three records from the `PERSONNEL` file, and if the persons are male, addresses them with a specified text:

```
§ READ(3) PERSONNEL BY NAME STARTING FROM 'A'
§ IF SEX ='M' DO
DEAR MR. §NAME
CONGRATULATIONS....
§ DOEND
§ LOOP
```

The resulting output text reads as follows:

```
DEAR MR. ALCOCK
CONGRATULATIONS....
DEAR MR. ALLAN
CONGRATULATIONS....
DEAR MR. AZOVEDA
CONGRATULATIONS....
```

5. Conditional Text Generation

Examples 2 and 4 above contain a macro text line which makes generation of text dependent on a specific condition:

```
§ IF #FIELD(#I) = ' ' ESCAPE BOTTOM END-IF    in example 2, and
§ IF SEX='M' DO                               in example 4
```

Using the Macro Feature in Natural ISPF

You can use the macro feature in Natural ISPF in any of the following ways:

- In a special Natural object called a macro object. Macro objects reside in Natural libraries and can be accessed as any other Natural object (specify `TYPE=MACRO`). Macro expansion is performed when the macro is executed; the macro output can be accessed for further handling in the user workpool facility. Additionally, macro objects can be referenced from various places within Natural ISPF.
- Inline macros in other sources (for example, PDS members, z/VSE members, LMS elements, Natural programs). The macros are executed as a result of certain function commands. The actual function is performed on the macro output, which can be seen as an intermediate file and is also written to the user workpool.

The following subsections describe each possibility in detail.

Macro Objects

You can access and maintain macro objects as any other Natural object if you specify `TYPE=MACRO` (see the subsection *Natural Objects* in the section *Common Objects* in the *Natural ISPF User's Guide*). However, please note the following when maintaining macro objects using the Editor:

- You must not use the `END` statement in macro objects;
- The `CHECK` command checks the processing statements and variables to be substituted in macro expansion for correct Natural syntax;

  Note: The `CHECK` command does not check that the text resulting from macro expansion is a valid Natural source. To do this, execute the macro object and store the resulting output from the user workpool as a Natural program (see the subsection *User Workpool* in the section *Common Objects* in the *Natural ISPF User's Guide*).

- The `CATALOG` and `STOW` commands compile the macro and create the "compiled" macro.

  You can `RUN` a macro source, and use compiled macro objects in any of the following ways:

  - `EXECUTE` a macro;
  - Issue the `COPY` or `SUBMIT` function command for a macro;
  - Use the Edit macro feature;
  - Reference a macro from another object using the `INCLUDE-MACRO` statement;
  - Execute a macro from applications outside of Natural ISPF;

- PLAY a macro to generate and execute a command script.

These uses are explained in the following subsections. Use of the INCLUDE-MACRO statement is described in the subsection ***Inline Macros***.

RUN / EXECUTE a Macro

When you issue the RUN command for a macro object, the macro source is executed, and the resulting output is written to the user workpool.

When you issue the EXECUTE command for a macro object, the compiled macro is executed, and the resulting output is written to the user workpool. If you are executing the macro from an application outside of Natural ISPF, the output is written to the source area (see also the subsection ***Using Macro Objects in other Natural Applications***).

The output of a macro object appears in the user workpool under the name of the macro and can be edited and saved (see the subsection *User Workpool* in the section *Common Objects* in the *Natural ISPF User's Guide*).

The following are two examples of macro objects. The first illustrates the use of variables to generate a Natural program, the second to generate job control lines.

Example 1: Using variables to generate a Natural program

This macro generates a part of a Natural program, which reads a specified number of records from a file in a logical sequence and displays the descriptor value and some other fields on the screen.

Macro definition:

```
§ DEFINE DATA LOCAL
§ 1 #FILE-NAME(A32)
§ 1 #KEY      (A32)
§ 1 #FIELD    (A32/5)
§ 1 #I        (N3)
§ END-DEFINE
§ *
§ SET CONTROL 'WL60C13B05/05F'
§ INPUT(AD=MIT'_') ' DISPLAY RECORD IN A FILE'
§     / ' FILE NAME  : ' #FILE-NAME
§     / ' KEY FIELD  : ' #KEY
§     / ' FIELD      : ' #FIELD(1)
§     / '            : ' #FIELD(2)
§     / '            : ' #FIELD(3)
§     / '            : ' #FIELD(4)
§     / '            : ' #FIELD(5)
READ(1) §#FILE-NAME!-VIEW BY §#KEY STARTING FROM  #VALUE
```

```
INPUT '§#KEY : '  20T ' ' §#KEY (AD=OI)
§ FOR #I = 1 TO 5
§ IF #FIELD(#I) = ' ' ESCAPE BOTTOM END-IF
 / ' §#FIELD(#I) : ' 20T ' ' §#FIELD(#I) (AD=MI)
§ END-FOR
END-READ
```

If you issue a `RUN` command from your Natural edit session, the macro is executed and you are prompted for input of the following fields:

```
+------------------------------------------------------+
!                                                      !
!  DISPLAY RECORD IN A FILE                            !
!  FILE NAME  :  ____________________________          !
!  KEY FIELD  :  ____________________________          !
!  FIELD      :  ____________________________          !
!             :  ____________________________          !
!             :  ____________________________          !
!             :  ____________________________          !
!             :  ____________________________          !
!                                                      !
!                                                      !
+------------------------------------------------------+
```

Assuming you specify the following values:

```
+------------------------------------------------------+
!                                                      !
!  DISPLAY RECORD IN A FILE                            !
!  FILE NAME  :  AUTOMOBILES_________________          !
!  KEY FIELD  :  MAKE________________________          !
!  FIELD      :  MODEL_______________________          !
!             :  COLOR_______________________          !
!             :  HORSEPOWER__________________          !
!             :  ____________________________          !
!             :  ____________________________          !
!                                                      !
!                                                      !
+------------------------------------------------------+
```

The variables are substituted with these values and the resulting output is written to the user workpool. You can see the output in the user workpool using the local command `OUTPUT`:

```
READ(1) AUTOMOBILES-VIEW BY  MAKE STARTING FROM  #VALUE
INPUT ' MAKE : '  20T ' ' MAKE (AD=OI)
 / ' MODEL : ' 20T ' ' MODEL (AD=MI)
 / ' COLOR : ' 20T ' ' COLOR (AD=MI)
 / ' HORSEPOWER : ' 20T ' ' HORSEPOWER (AD=MI)
END-READ
```

The resulting Natural program can be edited in the user workpool and saved (see the subsection *User Workpool* in the section *Common Objects* in the *Natural ISPF User's Guide*).

Example 2: Using variables to generate JCL lines.

The following macro object generates a job to perform a `SYSMAIN COPY` function, with the source and destination values given as variables:

```
§ RESET #JOBNAME(A8)
§ RESET #FD(N3) #FL(A8) #FF(N3)
§ RESET #TD(N3) #TL(A8) #TF(N3)
§ COMPRESS *INIT-USER 'SM' INTO #JOBNAME LEAVING NO SPACE
§ SET CONTROL 'WL60C6B005/010F'
§ INPUT  'ENTER PARAMETERS FOR LIBRARY COPY:'
§  /     'FROM:   DBID:' #FD 'FNR:' #FF 'LIB:' #FL
§  /     'TO  :   DBID:' #TD 'FNR:' #TF 'LIB:' #TL
//§#JOBNAME JOB JW0,MSGCLASS=X,CLASS=G,TIME=1400
//COPY EXEC PGM=NATBAT21,REGION=2000K,TIME=60,
// PARM=('DBID=9,FNR=33,FNAT=(,15),FSIZE=19',
//       'EJ=OFF,IM=D,ID='';'',MAINPR=1,INTENS=1')
//STEPLIB  DD  DISP=SHR,DSN=OPS.SYSF.V5.ADALOAD
//         DD  DISP=SHR,DSN=OPS.SYSF.PROD.LOAD
//DDCARD   DD  *
ADARUN DA=9,DE=3380,SVC=249
//CMPRINT  DD  SYSOUT=X
//CMPRT01  DD  SYSOUT=X
//CMWKF01  DD  DUMMY
//CMSYNIN  DD  *
LOGON SYSMAIN2
CMD C C * FM §#FL DBID §#FD FNR §#FF TO §#TL DBID §#TD FNR §#TF REP
FIN
```

If you issue the `RUN` command from your Natural edit session, the macro is executed and you are prompted for source and destination values in the following window:

```
ENTER PARAMETERS FOR LIBRARY COPY:
FROM:   DBID:      FNR:      LIB:
TO  :   DBID:      FNR:      LIB:
```

Assuming you enter `1` in the `FROM: DBID` and `FNR` fields, enter `2` in the `TO: DBID` and `FNR` fields, and enter `MYLIB` in both `LIB` fields, the Natural program is run and the output generated in the user workpool (use the local command `OUTPUT`):

```
//MBESM JOB JWO,MSGCLASS=X,CLASS=G,TIME=1400
//COPY EXEC PGM=NATBAT21,REGION=2000K,TIME=60,
// PARM=('DBID=9,FNR=33,FNAT=(,15),FSIZE=19',
//       'EJ=OFF,IM=D,ID='';'',MAINPR=1,INTENS=1')
//STEPLIB  DD  DISP=SHR,DSN=OPS.SYSF.V5.ADALOAD
//         DD  DISP=SHR,DSN=OPS.SYSF.PROD.LOAD
//DDCARD   DD  *
ADARUN DA=9,DE=3380,SVC=249
//CMPRINT  DD  SYSOUT=X
//CMPRT01  DD  SYSOUT=X
//CMWKF01  DD  DUMMY
//CMSYNIN  DD  *
LOGON SYSMAIN2
CMD C C * FM MYLIB DBID 1 FNR 1 TO MYLIB DBID 2 FNR 2 REP
FIN
/*
```

COPY / SUBMIT a Macro

Macro objects are separate objects in Natural ISPF and can be accessed directly by the `COPY` and `SUBMIT` function commands from any system screen using the object-type `MAC` in the command syntax. Available commands are:

Command	Parameter Syntax
COPY	*library*(*member*),*object-type object-parameters*,REP
SUBMIT	*library*(*member*),TARGET=*node-id*

Example: COPY 1

The function command:

```
COPY MAC MYLIB(MYPROG),NAT NEWLIB(NEWPROG)
```

executes macro object MYPROG in Natural library MYLIB and saves the output as Natural object NEWPROG in library NEWLIB.

Example: COPY 2

The macro program EXHEADER in Natural library SYSISPE dynamically creates a program header subsection with some information as to the edited object:

```
§ *
§ *  MACRO GENERATES A STANDARD PROGRAM HEADER FOR THE PROGRAM
§ *  BEING CURRENTLY EDITED
§ *
§ DEFINE DATA
§ LOCAL USING ISPN---L
§ LOCAL
§ 1 #OBJECT  (A2)
§ 1 #FUNCTION(A2)
§ 1 #DATA    (A200)
§ 1 #PROGRAM (A8)
§ 1 #TEXT    (A50/5)
§ 1 #I     (N2)
§ END-DEFINE
§ *
§ *  GET NATURAL SESSION DATA
§ *
§ CALLNAT 'ISP-U000' #OBJECT #FUNCTION #DATA
§ MOVE #DATA TO #SES-DATA-N
§ MOVE #MEMBER TO #PROGRAM
§ SET KEY PF3
§ SET CONTROL 'WL70C10B005/005F'
§ SET CONTROL 'Y45'
§ INPUT (AD=MIL'_')
§ WITH TEXT '----- PROGRAM HEADING INFORMATION  -------------'
§       'PROGRAM:' #PROGRAM (AD=OI) 'TYPE:' #OBJTYPE (AD=OI)
§       'LIBRARY:' #LIBRARY (AD=OI)
§ /     'PURPOSE:' #TEXT (1)
§ /     '        ' #TEXT (2)
§ /     '        ' #TEXT (3)
§ /     '        ' #TEXT (4)
§ /     '        ' #TEXT (5)
§ IF #TEXT(1) = ' '
§    REINPUT WITH TEXT 'PURPOSE IS REQUIRED'
§ END-IF
************************************************************************
* OBJECT : §#PROGRAM  DATE CREATED: §*DATD      BY: §*USER
* ---------------------------------------------------------------------
* PURPOSE:
§ FOR #I = 1 TO 5
§ IF #TEXT(#I) NE ' '
```

```
*         §#TEXT(#I)
§ END-IF
§ END-FOR
* ------------------------------------------------------------------------
* PROGRAM HISTORY
* DATE     USER-ID  REF-NO    DESCRIPTION
************************************************************************
*
§ IF #OBJTYPE NE 'C'
DEFINE DATA
§ IF #OBJTYPE NE 'N'
  GLOBAL USING ........
§   ELSE
  PARAMETER
§ END-IF
  LOCAL  USING ........
  LOCAL
END-DEFINE
*
* ----------
* Mainstream
* ----------
*
*
*
* --------------------
* Internal subroutines
* --------------------
*
END
§ END-IF
```

Note: For a specification of the subprogram `ISP-U000` and the Local Data Area `ISPN---L` referenced in this example, see the section *Application Programming Interface*.

If you start an editing session with a different object and issue the command:

```
COPY MAC SYSISPE(EXHEADER)
```

from the Editor command line, marking the place at which you want to see the header with the Editor line command `A`, the following window prompts you to input the required information:

```
      PROGRAM HEADING INFORMATION
PROGRAM: MYPROG   TYPE: P LIBRARY: SYSISPE
PURPOSE: _____________________________________________________
         _____________________________________________________
         _____________________________________________________
         _____________________________________________________
         _____________________________________________________
```

The header information includes the program name, type and library name and you can add a description of the program purpose. Assuming you enter: `Program to perform a function` and press ENTER, the following header is inserted at the specified place in your program:

```
************************************************************************
* OBJECT : MYPROG    DATE CREATED: 31.08.90      BY: MBE
* ----------------------------------------------------------------------
* PURPOSE:
*          PROGRAM TO PERFORM A FUNCTION
* ----------------------------------------------------------------------
* PROGRAM HISTORY
* DATE     USER-ID  REF-NO    DESCRIPTION
************************************************************************
*
DEFINE DATA
  GLOBAL USING ........
  LOCAL  USING ........
  LOCAL
END-DEFINE
*
* ----------
* Mainstream
* ----------
*
*
*
* --------------------
* Internal subroutines
* --------------------
*
END
```

Example: SUBMIT

The function command:

```
SUBMIT MAC MYLIB(MYPROG),TARGET=69
```

submits on Node 69 the output generated by the macro object `MYPROG` which resides in the Natural library `MYLIB`.

Note: (for z/OS only) In a similar way, with the function parameter `TYPE=TSO` or `TYPE=IDCAMS`, macro expansion can be used to pass the generated source lines to the TSO batch interface or to the `IDCAMS` utility.

Edit Macro

In the Edit macro field on the entry screens of some Natural ISPF objects (for example, PDS objects, Natural objects), you can specify the name of a macro object to be used as a model when editing a member. When starting the edit session with function command syntax, you must use the `MACRO=keyword` parameter.

The specified macro is executed, and the output appears in the edit area of the new member (the macro object referenced must be in the current library or in the library `SYSTEM` or `STEPLIB`).

Note: All lines generated by the macro are protected and cannot be modified.

Macro objects used as a model in this way are called "edit macros". They offer the following additional functions:

- Save variable values in the generated source;
- Define your own blocks of code in the generated source (user-edited blocks);
- Change the syntax of the information lines generated by the macro to match the syntax of the comments in the target language.

These functions are described in more detail below.

Save/Get Variable Values

You can save variable values specified during the execution of a macro object used for the Edit macro option (for example, if the macro prompts you for input values). If you wish to change any value and regenerate the macro output in the new source, use the `REGENERATE` command from the Editor command line, the other specified values remain in place.

A regeneration also takes place each time the member is selected for `EDIT` using the Edit macro option. If you select the member for `EDIT` without using the Edit macro option, the generated lines from the last generation are in place.

Variables used can be simple variables or arrays of any type. If you use arrays, you can reference variables in the following format:

```
#VAR(1)
#VAR(1:5)
```

Note: The notation `#VAR(*)` must not be used.

Variable values are saved using the `SAVE-DATA` statement after a `GET-DATA` clause in the macro object, as detailed below.

GET-DATA Statement

```
GET-DATA
     {USING local-name}
     {var-name}
END-GET
```

Note: Any line of this processing statement must be preceded by the macro character. The keyword `GET-DATA` must be the first string in the line.

Meaning of the variable parts:

Variable	Meaning
local-name	Name of local data area object
var-name	Variable name, optionally followed by an array definition, for example: `A(5), A(2:4), A(3,5), A(2:3,6), A(1:3,1:3,7:9)`

Function

When editing a new member with an Edit macro, the `GET-DATA` statement has no effect at all. However, when a subsequent `REGENERATE` command is executed, the `GET-DATA` statement restores variable field contents to the values of the last `SAVE-DATA` from the edited member. The field names are taken from the data areas, or are typed in explicitly.

Restrictions

1. The total number of fields is restricted to 128.
2. Length of field name must not exceed the following values:

   32 - Scalar field

   27 - One-dimensional array

   23 - Bi-dimensional array

   19 - 3-dimensional array
3. When using data areas:
   - only fields of level one are taken;
   - fields can be scalar or array (any dimension).
4. The notation *#var-name*`(*)` is not valid.

Examples

```
§ GET-DATA
§   #ALFA
§   #NUM
§   #VEC(3)
§   #VEC(2:5)
§   #VEC(4,6:7)
§   #VEC(4,4,4)
§ END-GET
```

```
§ GET-DATA USING G-LOCAL END-GET
```

```
§ GET-DATA
§   #MY-VAR
§   USING G-LOCAL
§ END-GET
```

SAVE-DATA Statement

```
SAVE-DATA  ALL
```

or:

```
SAVE-DATA
 {USING local-name}
 {var-name}
END-SAVE
```

Note: Any line of this processing statement must be preceded by the macro character. The keyword `SAVE-DATA` must be the first string in the line.

Meaning of the variable parts:

Variable	Meaning
local-name	Name of local data area object
var-name	Variable name, optionally followed by an array definition, for example: `A(5), A(2:4), A(3,5), A(2:3,6), A(1:3,1:3,7:9)`

Function

The SAVE-DATA statement saves the variable contents in the generated source. These values can later be retrieved using the GET-DATA statement. The SAVE-DATA ALL option refers to the variable list of the previous GET-DATA statement. The SAVE-DATA ALL option is valid only if the GET-DATA statement is contained in the same macro object. The field values are taken from the data areas, or are typed in explicitly.

Note: The GET-DATA clause has no effect when the Edit macro option is used to create a new member. However, when editing an existing member, or when issuing the REGENERATE command, it reads the variable data for the previous execution, the SAVE-DATA statement writes the data to the generated source.

Restrictions

The same restrictions apply as for the GET-DATA statement (see **above**).

Examples

```
§ SAVE-DATA ALL
```

```
§ SAVE-DATA USING G-LOCAL END-SAVE
```

```
§ SAVE-DATA
§   #MY-VAR
§   USING G-LOCAL
§ END SAVE ↩
```

Example: Save Variable Values

The following macro object named EXMOD generates a program to invoke any application:

```
0010 § DEFINE DATA LOCAL
....
0040 § 1 #M-LIB  (A8)
0050 § 1 #M-START(A8)
0060 § END-DEFINE
0070 § *
0080 § GET-DATA
0090 §    #M-LIB
0100 §    #M-START
0110 § END-GET
0120 § *
0130 § INPUT (AD=IM) 'APPLICATION :' #M-LIB /
0140 §               'STARTPROGRAM:' #M-START
0150 § IF #M-LIB EQ ' ' STOP END-IF
0160 § SAVE-DATA ALL
```

```
0170 DEFINE DATA
0180 LOCAL
0190 01 DUMMY  (A1)
0200 § BEGIN-BLOCK 'DATA'
0210 ** HERE YOU CAN DEFINE YOUR OWN FIELDS
0220 § END-BLOCK
0230 END-DEFINE
0240 § BEGIN-BLOCK 'START'
0250 ** HERE YOU CAN ENTER YOUR OWN STATEMENTS
0260 § END-BLOCK
0270 § IF #M-START NE ' '
0280 STACK TOP COMMAND '§#M-START'
0290 § END-IF
0300 STACK TOP COMMAND 'LOGON §#M-LIB'
0310 STACK TOP COMMAND 'SETUP * SPF'
0320 *
0330 END
```

If you specify this macro as edit macro when starting an edit session with new Natural member `MYPROG` in the library `MYLIB` using the command:

```
EDIT NAT MYLIB(MYPROG) MACRO=EXMOD
```

you are prompted for the input values:

```
APPLICATION
STARTPROGRAM
```

The following output is written to the edit session using input values `MYAPPL` for `APPLICATION` and `STARTUP` for `PROGRAM`:

```
=P0001 ***M     GENERATED USING:EXMOD
=P0010 ***MSV #M-LIB = MYAPPL
=P0020 ***MSV #M-START = STARTUP
=P0030 DEFINE DATA
=P0040 LOCAL
=P0050 01 DUMMY  (A1)
=P0060 ***MBB DATA
000070 ** HERE YOU CAN DEFINE YOUR OWN FIELDS
=P0080 ***MBE
=P0090 END-DEFINE
=P0100 ***MBB START
000110 ** HERE YOU CAN ENTER YOUR OWN STATEMENTS
=P0120 ***MBE
=P0130 STACK TOP COMMAND 'STARTUP'
=P0140 STACK TOP COMMAND 'LOGON MYAPPL'
=P0150 STACK TOP COMMAND 'SETUP * SPF'
=P0160 *
=P0170 END
```

If you now enter the command:

```
REGENERATE
```

in the Editor command line, the macro program is regenerated and the prompt for application and start program reappears with the values last specified. You can modify any value and press ENTER to load the new output in your edit session. Any unchanged variable retains its old value.

User-Edited Blocks in Generated Source

You can define your own blocks of code in the source generated by the macro object executed using the Edit macro option. Each block starts with a line signalling the beginning of the block, indicating a string or variable (`BEGIN-BLOCK` statement). The block closes with a line signalling the end of the block (`END-BLOCK` statement). The corresponding syntax in the macro object is:

```
BEGIN-BLOCK block-identifier
        text-line ...
END-BLOCK
```

Note: The lines beginning with the keywords `BEGIN-BLOCK` and `END-BLOCK` must be preceded by the macro character. The keywords `BEGIN-BLOCK` and `END-BLOCK` must be the first string in the respective lines.

Meaning of the variables:

Variable	Meaning
block-identifier	Can either be an alphanumeric constant or a variable of format A.
text-line	A line used to initialize the block when a new source is generated.

Function

After macro execution, you can write your own lines of code at the designated place when editing the output generated by the Edit macro option. The next time you start an edit session with the member and you wish to regenerate the source using the Edit macro option, your user-edited blocks remain intact.

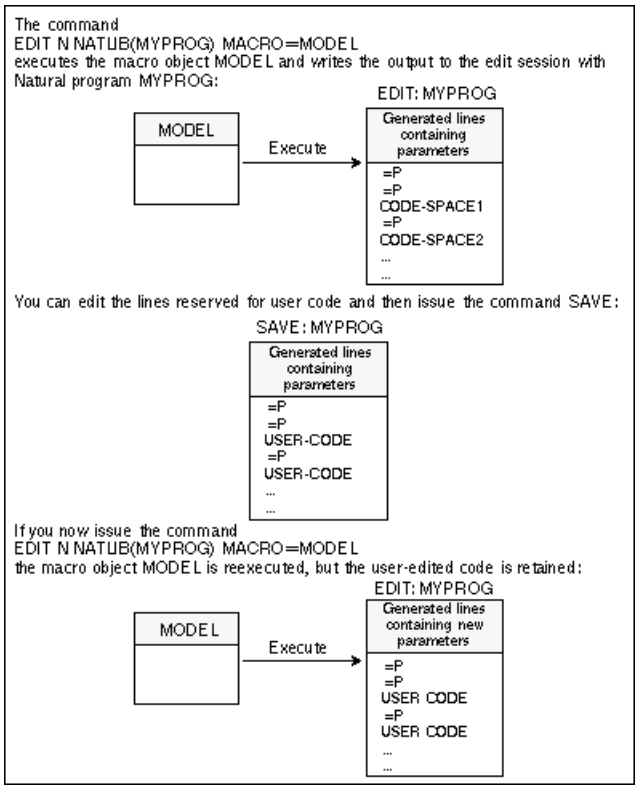
Restrictions

- The block-identifier must not exceed 8 characters and must be unique within the scope of the generated source (that is, different blocks must carry different identifiers).
- No macro processing statements are allowed within a user-edited block.

Example

```
0200 § BEGIN-BLOCK 'string' (or: #variable)
0210 <text lines to initialize the block for the first time>
0220 § END-BLOCK
```

The following figure illustrates the sequence of events described above:



Example: User-Defined Blocks in Generated Source

Using the example macro object `EXMOD` illustrated in the above example of saving variables, consider the text generated by the command:

```
EDIT NAT MYLIB(MYPROG) MACRO=EXMOD
```

specifying `MYAPPL` and `STARTUP` as input values for the prompted application and start program:

```
=P0001 ***M     GENERATED USING:EXMOD
=P0010 ***MSV #M-LIB = MYAPPL
=P0020 ***MSV #M-START = STARTUP
=P0030 DEFINE DATA
=P0040 LOCAL
=P0050 01 DUMMY  (A1)
=P0060 ***MBB DATA
000070 ** HERE YOU CAN DEFINE YOUR OWN FIELDS
=P0080 ***MBE
=P0090 END-DEFINE
=P0100 ***MBB START
000110 ** HERE YOU CAN ENTER YOUR OWN STATEMENTS
=P0120 ***MBE
=P0130 STACK TOP COMMAND 'STARTUP'
=P0140 STACK TOP COMMAND 'LOGON MYAPPL'
=P0150 STACK TOP COMMAND 'SETUP * SPF'
=P0160 *
=P0170 END
```

You can now define your own lines of code in the lines containing the comment `HERE YOU CAN DEFINE YOUR OWN FIELDS/STATEMENTS`, for example:

```
=P0001 ***M     GENERATED USING:EXMOD
=P0010 ***MSV #M-LIB = MYAPPL
=P0020 ***MSV #M-START = STARTUP
=P0030 DEFINE DATA
=P0040 LOCAL
=P0050 01 DUMMY  (A1)
=P0060 ***MBB DATA
000070 01 #STARTDATA (A10) INIT <'INIT'>
=P0080 ***MBE
=P0090 END-DEFINE
=P0100 ***MBB START
000110 SET CONTROL 'MT'
000120 STACK TOP DATA #STARTDATA
=P0130 ***MBE
=P0130 STACK TOP COMMAND 'STARTUP'
=P0130 STACK TOP COMMAND 'LOGON MYAPPL'
=P0160 STACK TOP COMMAND 'SETUP * SPF'
=P0170 *
=P0180 END
```

You can save this source using the SAVE command. If you now start an edit session with this member, regenerating the source by specifying the macro object EXMOD in the command:

```
EDIT NAT MYLIB(MYPROG) MACRO=EXMOD
```

and specifying other values in the APPLICATION and STARTPROGRAM prompt, for example YOURAPPL and START, the following output is loaded in the edit area:

```
=P0001 ***M     GENERATED USING:EXMOD
=P0010 ***MSV #M-LIB = YOURAPPL
=P0020 ***MSV #M-START = START
=P0030 DEFINE DATA
=P0040 LOCAL
=P0050 01 DUMMY  (A1)
=P0060 ***MBB DATA
000070 01 #STARTDATA (A10) INIT <'INIT'>
=P0080 ***MBE
=P0090 END-DEFINE
=P0100 ***MBB START
000110 SET CONTROL 'MT'
000120 STACK TOP DATA #STARTDATA
=P0130 ***MBE
=P0140 STACK TOP COMMAND 'START'
=P0150 STACK TOP COMMAND 'LOGON YOURAPPL'
=P0160 STACK TOP COMMAND 'SETUP * SPF'
=P0170 *
=P0180 END
```

Change Syntax Format

Source lines generated using the Edit macro option can be adapted to match the syntax of the target language. This is done using the DATA-FORMAT statement as the first executable macro statement in the macro object:

```
DATA-FORMAT=[sssssss] [yyy]
```

Note: This statement can only occur as the first executable macro statement. It must be preceded by the macro character, and the keyword DATA-FORMAT must be the first string in the line.

Meaning of the variables:

Variable	Meaning
sssssss	Prefix string of up to 7 characters. The default is ***M.
yyy	Suffix string of up to 3 characters. By default, this string is empty.

Notes:

1. The notation MODEL-DATA-FORMAT is also valid. The equal sign (=) is optional and can be omitted if the keyword DATA-FORMAT is followed by at least one separating blank, followed by the prefix and/or suffix strings.
2. The prefix and/or suffix string can optionally be enclosed in apostrophes ('). This notation is required if the specified string contains one or more blanks, or if it is a prefix string ending with a blank character (that is, if the prefix must be separated from remaining text by a blank).

Function

In some cases, the invoked macro writes its own data into the source area (for example, saved variables). The DATA-FORMAT statement provides a prefix and suffix for that data. This definition must reflect a comment in the target language.

Restrictions

1. The whole DATA-FORMAT statement cannot exceed one line.
2. The prefix or suffix string cannot contain commas or apostrophes. If any string contains blanks, the whole string must be enclosed in apostrophes.

Examples

```
§ DATA-FORMAT ******C

§ DATA-FORMAT=/*,*/        /* for PL1

§ DATA-FORMAT '/REMA '     /* for BS2000 job control
```

An example macro using the DATA-FORMAT statement follows on the next page.

Example: Change Syntax Format

The following macro object generates a job to perform a tape scan on a specified volume. The `DATA-FORMAT` statement specifies JCL as the syntax format for the generated source:

```
0010 § DATA-FORMAT //*
0020 § RESET #VOL(A6)
0030 § GET-DATA
0040 §  #VOL
0050 § END-GET
0060 § SET CONTROL 'WL60C10B005/005F'
0070 § INPUT (AD=MI'_') WITH TEXT 'ENTER VOLSER FOR TAPESCAN'
0080 §  / ' VOLSER            ' #VOL
0090 § SAVE-DATA
0100 //JWOTP12  JOB  JWO,CLASS=1,MSGCLASS=X,REGION=2500K
0110 //SCAN     EXEC TAPESCAN,TAPE=§#VOL
```

The following source is generated using the command:

```
EDIT NAT MYLIB(TAPESC) MACRO=EXJCL
```

and specifying volume `COM811` in the prompt window that appears during macro execution:

```
=P0001 //*      GENERATED USING:EXJCL
=P0010 //*FR //*
=P0020 //*SV #VOL = COM811
=P0030 //JWOTP12  JOB  JWO,CLASS=1,MSGCLASS=X,REGION=2500K
=P0040 //SCAN     EXEC TAPESCAN,TAPE=COM811
```

Using Macro Objects in Other Natural Applications

When you wish to execute macro objects from a Natural application or program outside Natural ISPF, the generated output is written to the source area, where it can be edited using the standard Natural program editor, and run in a production environment. A more detailed description follows in the subsections below.

Macro objects are invoked from other applications using the statement:

```
FETCH RETURN 'name' parameters
```

where *name* is the name of the macro to be invoked and *parameters* the parameters to be passed to the macro as required.

Note: The macro must be a cataloged object in the library `SYSTEM` or `STEPLIB`.

Generating Natural Code in Natural Applications

Macro objects to be invoked in Natural applications outside Natural ISPF must carry their own generation parameters. This is done by coding an appropriate `SET-MACRO` statement in the macro as follows:

```
SET-MACRO
  parameter-definition ...
END-SET
```

where *parameter-definition* takes the following format:

```
NAME   = object-name
SMODE  = {S}
         {R}
TYPE   = {P}}
       = {C}
       = {S}
       = {N}
       = {A}
       = {L}
       = {M}
       = {G}
       = {H}
       = {T}
```

Note: Any line of this processing statement must be preceded by the macro character. The keyword `SET-MACRO` must be the first string in the respective line.

Meaning of variable:

Variable	Meaning
object-name	Name given to the generated code. It must not exceed 8 characters and can be specified as an alphanumeric constant or the content of an alphanumeric variable.

The values of the other keywords refer to the type and structure of the generated code.

Function

The `SET-MACRO` statement defines the macro generation parameters described under the above syntax. Note that if invoked using the Edit macro option, the `NAME` parameter specified using the `SET-MACRO` statement in the invoked macro is overridden by the name specified in the Edit macro call.

Examples

```
§ SET-MACRO NAME='MY-PROG' END-SET
```

```
§ SET-MACRO
§ NAME=#INPUT-PROG
§ TYPE=N
§ SMODE=S
§ END-SET ↩
```

An example that demonstrates the use of this statement can be found in the Example Library, objects MAC-RUNZ and MAC-RUNP. Executing the program MAC-RUNP causes a Natural object to be generated, cataloged and executed.

Using Macro Objects with GET-DATA / SAVE-DATA Statements

If the invoked macro uses GET-DATA/SAVE-DATA statements (see the subsection ***Edit Macro***), the Natural subprogram ISP--RVU must be called. This subprogram extracts the data from the source area and clears the source area. It must be called before the macro is executed, and expects the output of the last execution of the macro in the source area. Additionally, the subprogram ISP--RVU provided has the following parameters:

Name	Type	I/O	Meaning	
#MACRO	(A8)	I/O	'-empty-'	If source area is empty when called.
			' '	No appropriate text found in source area.
			'name'	Name of the macro program which generated the source
#ERROR-CODE	(N3)	O	Non-zero if error occurred.	
#ERROR-TEXT	(A75)	O	Explanation of error.	

If you wish to use this feature, the following programs must be copied from the Natural ISPF user exit library to your application or to a valid STEPLIB:

```
ISP--RVU
ISP--RVN
```

PLAY a Macro

PLAY Command

The Natural ISPF function command `PLAY` allows you to execute sequences of Natural ISPF commands stored as any of a number of Natural ISPF objects (PDS member, Natural object, z/VSE member, LMS element, workpool output, or, as explained below, as a macro object). For details on the `PLAY` feature, see the subsection *Executing Command Scripts* in the section *Useful Features* in the *Natural ISPF User's Guide*.

Generate Command Script

A command script can be generated by a macro, allowing scripts to be created and played dynamically. This can be done with the following syntax, valid from any system screen:

```
PLAY MAC library(member)
```

Here, the member must be a cataloged Natural object of type `O` (macro) or of type `P` (program). For special considerations applying to type `P`, see the subsection ***Splitting Macro Objects into Modules***.

Example

For example, executing the following macro with the `PLAY` command generates a prompt for a `CHANGE` command to be used on a member, with a choice of a `STOW` or `SAVE` command after the change is made:

```
§ RESET #MEMBER(A8) #FROM(A16) #TO(A16) #STOW(A1)
§ INPUT(AD=MI) 'Change' #FROM 'To' #TO 'in member' #MEMBER
           / 'Stow?' #STOW(A1)
EDIT NAT §#MEMBER
CHANGE '§#FROM' '§#TO' all
§ IF #STOW NE ' '
STOW
§ ELSE
SAVE
§ END-IF
END
```

For another example, see member `VERIFY` in the Example Library. This macro generates a script that verifies whether or not Natural ISPF has been installed correctly in your environment.

Inline Macros

Apart from macro objects, other sources, such as PDS members, Natural programs, CA Panvalet members, etc., can use the macro feature by including inline macros. Inline macros are processing statements and variables included in a member. As a result of certain function commands, the member is checked for macro statements, and if any are found, the member is run as a macro object: the output is held in an intermediate file written to the user workpool. The invoked function is then performed on the intermediate file.

Inline macros also allow the use of a special `INCLUDE-MACRO` statement that can invoke a macro object and include its output in the member. The `INCLUDE-MACRO` statement takes the following format:

```
INCLUDE-MACRO name [parameter]
```

Note: This statement must be preceded by the macro character.

Meaning of the variables:

Variable	Meaning
name	Identifies the compiled macro to be included. It can be an alphanumeric constant or variable and must not exceed 8 characters in length.
parameter	Parameters that can be sent to the macro to be received by means of the `INPUT` statement.

Note: The macro object invoked by the `INCLUDE-MACRO` statement must be a cataloged (`CATALOG` or `STOW` command) object in the current Natural library, or library `SYSTEM` or `STEPLIB`.

The function commands that perform macro expansion of inline macros and `INCLUDE-MACRO` statements are:

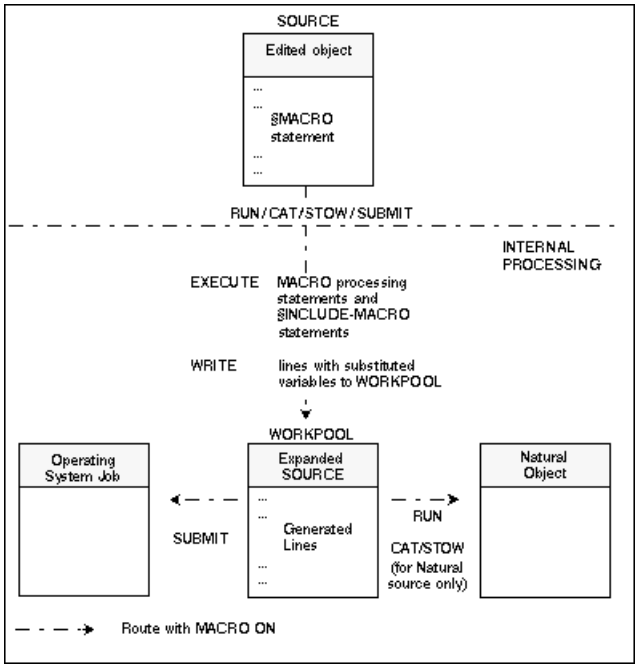
- For Natural programs: `CHECK`, `RUN`, `CATALOG`, `STOW`, `SUBMIT`;
- For other sources (PDS, Librarian members, etc.): `SUBMIT`.

Note: The macro facility must be enabled either with the command `MACRO ON` or by setting the `MACRO EXPAND` option in the user defaults of your user profile to `Y`.

Important: When using inline macros in any non-Natural source, you must specify the Natural programming mode before starting an edit session or issuing a `SUBMIT` command. You do this via the `MACRO SMODE` setting in the user defaults of your user profile (see the section *Profile Maintenance* in the *Natural ISPF User's Guide*). If no mode is specified in your user profile, the default is the mode defined by the system administrator.

Tip: Instead of submitting non-Natural members containing inline macros, a better performance can be achieved by copying the member as a macro object to a Natural library, compiling it, and then submitting it.

The following figure illustrates the use of inline macros:



Note: If the macro facility is disabled (for example with the `MACRO OFF` session command), the function is executed directly on the source.

Example: Inline Macros in a Natural Program

Below is an example of a Natural program which contains an `INCLUDE-MACRO` statement. The program reads specified records from the file `AUTOMOBILES`:

```
DEFINE DATA LOCAL
1 AUTOMOBILES-VIEW VIEW OF AUTOMOBILES
  2 MAKE
  2 MODEL
  2 COLOR
1 #VALUE(A20)
END-DEFINE
*
INPUT #VALUE (AD=MIT'_')
§ INCLUDE-MACRO 'EXF1' 'AUTOMOBILES' 'MAKE' 'MODEL' 'COLOR'
END
```

Below is the macro object `EXF1` called by the `INCLUDE-MACRO` statement:

```
§ DEFINE DATA
§ LOCAL USING EXFL
§ LOCAL
§ 1 #I(N3)
§ END-DEFINE
§ *
§ DEFINE WINDOW WIND1 SIZE 13 * 60
§   BASE 10/10
§   CONTROL SCREEN
*
§ SET WINDOW 'WIND1'
§ INPUT(AD=MIT'_') ' DISPLAY RECORD IN A FILE'
§     / ' FILE NAME  : ' #FILE-NAME   0007 JWO      94-12-14 18:02
§     / ' KEY FIELD  : ' #KEY
§     / ' FIELD      : ' #FIELD(1) 2  0006 JWO      94-02-18 11:02
§     / '            : ' #FIELD(2)
§     / '            : ' #FIELD(3)
§     / '            : ' #FIELD(4)
§     / '            : ' #FIELD(5)
§ SET WINDOW OFF
READ(1) §#FILE-NAME BY  §#KEY STARTING FROM  #VALUE
INPUT ' §#KEY : '  20T ' ' §#KEY (AD=OI)
§ FOR #I = 1 TO 5
§ IF #FIELD(#I) = ' ' ESCAPE BOTTOM END-IF
 / ' §#FIELD(#I) : ' 20T ' ' §#FIELD(#I) (AD=OD)
§ END-FOR
END-READ
```

If you issue the `RUN` command from your Natural edit session, you are prompted for the variable `VALUE` which corresponds to the starting value of the records to be read. If you enter the required value (for example, `FERRARI`), you are prompted for the fields `MAKE`, `MODEL` and `COLOR`. Type in the required values and press ENTER. The output of the program is written to the user workpool under the name `##INLINE`.

```
DEFINE DATA LOCAL /* L0060
1 AUTOMOBILES-VIEW VIEW OF AUTOMOBILES /* L0070
  2 MAKE /* L0080
  2 MODEL /* L0090
  2 COLOR /* L0100
1 #VALUE(A20) /* L0110
END-DEFINE /* L0120
INPUT #VALUE (AD=MIT'_') /* L0150
READ(1) AUTOMOBILES-VIEW BY  MAKE STARTING FROM  #VALUE
INPUT ' MAKE : '  20T ' ' MAKE (AD=OI)
 / ' MODEL : ' 20T ' ' MODEL (AD=MI)
 / ' COLOR : ' 20T ' ' COLOR (AD=MI)
END-READ
END /* L0170
```

Example: Inline Macros in a PDS Member

The macro object used as an example for the substitution of variables in JCL lines described in the subsection ***Macro Objects*** could also be a PDS member: the job performs a `SYSMAIN COPY` function, with the source and destination values given as variables:

```
§ RESET #JOBNAME(A8)
§ RESET #FD(N3) #FL(A8) #FF(N3)
§ RESET #TD(N3) #TL(A8) #TF(N3)
§ COMPRESS *INIT-USER 'SM' INTO #JOBNAME LEAVING NO SPACE
§ INPUT  'ENTER PARAMETERS FOR LIBRARY COPY:'
§  /     'FROM:   DBID:' #FD 'FNR:' #FF 'LIB:' #FL
§  /     'TO  :   DBID:' #TD 'FNR:' #TF 'LIB:' #TL
//§#JOBNAME JOB JWO,MSGCLASS=X,CLASS=G,TIME=1400
//COPY EXEC PGM=NATBAT21,REGION=2000K,TIME=60,
// PARM=('DBID=9,FNR=33,FNAT=(,15),FSIZE=19',
//       'EJ=OFF,IM=D,ID='';'',MAINPR=1,INTENS=1')
//STEPLIB  DD  DISP=SHR,DSN=OPS.SYSF.V5.ADALOAD
//         DD  DISP=SHR,DSN=OPS.SYSF.PROD.LOAD
//DDCARD   DD  *
ADARUN DA=9,DE=3380,SVC=249
//CMPRINT  DD  SYSOUT=X
//CMPRT01  DD  SYSOUT=X
//CMWKF01  DD  DUMMY
//CMSYNIN  DD  *
LOGON SYSMAIN2
CMD C C * FM §#FL DBID §#FD FNR §#FF TO §#TL DBID §#TD FNR §#TF REP
FIN
```

If you issue the `SUBMIT` command from your PDS edit session, the macro processing statements are executed and you are prompted for source and destination values in the following window:

```
  ENTER PARAMETERS FOR LIBRARY COPY:
  FROM:   DBID:      FNR:      LIB:
  TO  :   DBID:      FNR:      LIB:
```

Assuming you enter `1` in the `FROM: DBID` and `FNR` fields, enter `2` in the `TO: DBID` and `FNR` fields, and enter `MYLIB` in both `LIB` fields, the JCL lines are generated and the job is submitted to the operating system. You can access and maintain the generated JCL in the user workpool under the name `##SUBMIT`:

```
//MBESM JOB JWO,MSGCLASS=X,CLASS=G,TIME=1400
//COPY EXEC PGM=NATBAT21,REGION=2000K,TIME=60,
// PARM=('DBID=9,FNR=33,FNAT=(,15),FSIZE=19',
//       'EJ=OFF,IM=D,ID='';'',MAINPR=1,INTENS=1')
//STEPLIB  DD  DISP=SHR,DSN=OPS.SYSF.V5.ADALOAD
//         DD  DISP=SHR,DSN=OPS.SYSF.PROD.LOAD
//DDCARD   DD  *
ADARUN DA=9,DE=3380,SVC=249
//CMPRINT  DD  SYSOUT=X
//CMPRT01  DD  SYSOUT=X
//CMWKF01  DD  DUMMY
//CMSYNIN  DD  *
LOGON SYSMAIN2
CMD C C * FM 1 DBID 1 FNR 1 TO 2 DBID 2 FNR 2 REP
FIN
/*
```

Splitting Macro Objects into Modules

When you are writing a macro designed to generate larger amounts of data (for example, large batch jobs), certain technical limitations (for example, `ESIZE` restrictions) are encountered. To avoid these problems, the macro object should be split into several modules to create entire macro applications.

Because a cataloged Natural object of type P (program) can also be used as a macro, the object can also be addressed from Natural ISPF with the commands `COPY MAC`, `PLAY MAC` or `SUBMIT MAC`. The `FETCH RETURN` statement can then be used to branch from such a program to a real macro object, that generates JCL, for example. This statement works for macro objects in the same way as it does for Natural programs: the text lines generated by several macro objects called in succession, are simply accumulated in the User Workpool.

Example

The following program `IG-----P` can be executed with the command:

```
SUBMIT MAC PROB-DE1(IG-----P)
```

It executes a screen dialog and then addresses various other Natural objects. Some of these are programs (`IGDOCM-P`, `IGDNEW-P`), which perform online actions, but others are macro objects (`IGJOBC-Z`, `IGIEBC-Z`), that generate JCL lines.

```
000010 DEFINE DATA
000020             GLOBAL USING  IF-----G
000030             LOCAL  USING  XXCTIT-A
000040             LOCAL
------ ---------------------------------------------- 740 line(s) not displayed
007450   INPUT USING MAP 'IGINP-1M'
007460   SET CONTROL 'WB'
007470   DECIDE ON EVERY VALUE OF ##TYPE(#K)
007480     VALUE 'D'
007490       FETCH RETURN 'IGDOCM-P'         /* Create documentation member
007500     VALUE 'H'
007510       FETCH RETURN 'IGDNEW-P' ##PARM(#K)  /* Update the news member
007520     VALUE 'X'                          /* user defined
007530       FETCH RETURN ##PROGRAM(#K) ##PARM(#K)
007540     VALUE 'S', 'U' , 'B' , 'I' , 'Z'
007550       IF #JOB-CREATED EQ FALSE
007560         FETCH RETURN 'IGJOBC-Z'         /* Job-card
007570         MOVE TRUE TO #JOB-CREATED
007580       END-IF
007590     VALUE 'I'
007600       FETCH RETURN 'IGIEBC-Z' ##PARM(#K)  /* IEBCOPY
```

Saving Macro Output in the User Workpool

The output of objects that use the Natural ISPF macro facility is written to the user workpool at execution time. This subsection summarizes which commands can be used for which object types to write output to the workpool and under what name the output appears in the workpool:

Object Type	Command	Name of Output in Workpool
Macro	RUN	*macro-name*
	EXECUTE	*macro-name*
	PLAY	##PLAY
Natural program with inline macros	STOW/CAT/RUN	##INLINE
Macro and other objects with inline macros	SUBMIT	##SUBMIT

Note: Workpool files are intermediate files only. If you wish to keep source that was generated in the workpool, it is strongly recommended that you store it as another object elsewhere in Natural ISPF. See the subsection *Saving Output* in the section *Common Objects* of the *Natural ISPF User's Guide*).

2 Incore Database

Overview

Traditionally, data processing applications mainly handle fields and files. With the technological advances in the field of data processing, however, the need for manipulating more complex data structures is arising on a large scale. Applications must be able to support the integration of texts, tables, images, graphics, etc.

The Natural ISPF Incore Database facility enables the Natural programmer to maintain complex data structures in the user memory, and to perform complex actions on these structures. Using the Incore Database, you can handle texts, reports, files and tables using the Natural language.

Advantages of Incore Files

Using incore files in application development has several advantages:

- You can integrate Software AG Editor functions in Natural programs, enabling flexible manipulation of your incore files;
- An incore file is a temporary workfile of unlimited space running in memory;
- You can have your personal environment to test and prototype your applications away from your site's database administration activities. After prototyping, no further source changes are required to access a real database;
- If you wish to keep the contents of an incore file permanently, you can write them to a container file. You can retrieve the incore file later, thus avoiding the need to regenerate the data from your programs.

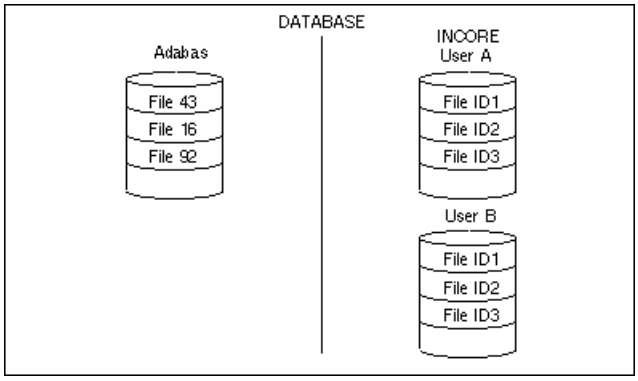
Functions of the Incore Database Facility

The Incore Database facility allows you to perform the following:

- Define an incore file structure using Natural DDMs;
- Dynamically create and delete incore files;
- Manipulate the data in an incore file using the Natural DML or the `WRITE` *`reportn`* statement;
- Invoke the `INCORE` processor (with `CALLNAT` interface) to perform operations on an incore file as a whole, for example:
  - `BROWSE` an incore file (for example, one which contains a report)
  - `EDIT` an incore file (for example, one which contains text)
  - `STORE` incore files as a whole into a container file and retrieve them later

Incore database files can be handled like Adabas files with some restrictions on field types, see the subsection ***Defining Fields of an Incore File***. Incore files are not identified by a file number but by an identifier, this allows multiple copies of a file created with the same DDM to be in the

database. Incore files are temporary files which cannot be shared by users, each file belongs to only one user. The following figure illustrates this concept:



Incore files are allocated dynamically and stored in memory and, if required, swapped to disk (in fact, the files are stored in the Editor Buffer Pool). Incore files provide "unlimited memory" to a user. The Incore Database can be used in an online environment as well as in Batch. If incore files are to be kept permanently, they can be stored in the `CONTAINER` (usually an Adabas file) for later retrieval.

Note: The theoretical maximum for incore files per user is 50. For this reason it is recommended to delete unused incore files, especially when the error message NAT6896 (Session does not exist) occurs.

Defining Fields of an Incore File

You define the fields of an incore file by defining a Natural DDM. This can be done with the Natural utility `SYSDDM` or with Software AG's data dictionary Predict. In Predict, an incore database is defined exactly as an Adabas database. The DDM is recognized as an incore file by using a special DBID (see the the *Natural ISPF Installation* documentation or ask your administrator). The file number (FNR) must be a value lower than 191.

Some Information about Fields

- Incore DDMs can include fields of type: Alphanumeric, Numeric, Binary and Packed. Fields can be defined as MU, and can be defined as descriptors. The following types are not supported: Sub, Super, Phonetic and Hyperdescriptors.
- Periodic groups are now supported, but internally handled as a series of multiple value fields and therefore may not themselves contain multiple value fields.
- In addition to these fields, you can define a control pseudo-field (that is, no real database field):

```
RN RECORD-NUMBER (N7)
```

This field is used to access a physical record number within an incore file.

- The number of fields of an incore file (including multiple occurrences) is limited to 66 (for exceptions to this limitation, see below).

Some Information about Record Size

- The record size of an incore file manipulated using Natural DML or using the `CALLNAT INCORE` interface (not with `CONTAINER` actions) is limited to 4000 - *n* bytes, where *n* is the number of fields within the file.
- If an incore file is to be stored in / retrieved from the Natural ISPF Container File, its record size is limited to 3896 - *n* bytes.
- However, you can define and use an incore file with more than 66 fields, if the record size of the file does not require the maximum sizes mentioned above. To be precise, for each additional field over 66, the maximum record size must be decreased by 28 bytes.
- If a view definition contains the pseudo field `RECORD-NUMBER`, this field can be ignored with respect to the above limitations of record sizes and number of fields.

Identifying an Incore File

Each incore file has an identifier, Format A8. This identifier must be supplied (but can be supplied implicitly) with every Natural statement and incore processor call that refers to the incore file. The identifier notation is:

```
IDENTIFIER =const/var
```

If the identifier is omitted, the default is set implicitly by prefix `L` plus the three-digit FNR. For example, if the incore file to be referenced has file number 12, the default identifier is L012.

Example

The following Natural DML lines read an incore file with identifier `PER01` and display fields `NAME`, `AGE`, `CITY`:

```
....
0080 READ INCORE-PERSONNEL IDENTIFIER = 'PER01'
0090   DISPLAY NAME  AGE CITY
....
```

The following Natural statements browse an incore file with identifier `PER02` using an ISPF-like user interface:

```
...
0120 MOVE 'BROWSE' TO ACTION
0130 MOVE 'PER02' TO FILE-IDENTIFIER
0140 CALLNAT INCORE USING INCORE-CTL INCORE-DATA
0150 IF ERROR-CODE > 0
....
```

Creating an Incore File

After an incore DDM has been defined, incore files can be created in several ways:

- Implicitly with the first store statement.
- Explicitly using a `CALLNAT` interface statement. Using this method, file-internal parameters can be controlled and null files can be created, since no initial `STORE` is required. This interface is described in more detail in a later subsection.
- By `WRITE`/`DISPLAY` report statements. With this mechanism, reports can be stored and manipulated in the incore database.

Creating an Incore File Implicitly

An incore file can be created implicitly with the first `STORE` statement. The actual record format is then set by the fields that are specified with the `STORE` statement.

If the file contains multiple value fields or periodic groups, the first `STORE` statement must specify all occurrences that are to be used with this file. Subsequent `STORE` statements can specify selected occurrences as long as they do not specify an occurrence greater than the highest occurrence from the first `STORE` statement.

Example

The Natural DML lines

```
....
0080 STORE INCORE-PERSONNEL IDENTIFIER = 'PER01'
0090   WITH NAME = 'SMITH'  AGE = 20   CITY = 'NY'
....
```

create a new incore file with identifier `PER01` and fields `NAME`, `AGE`, `CITY`. One record is stored in the file, as specified by the field values.

When creating an incore file implicitly, no additional parameters for creating the file can be defined, the default values will be taken by incore database. If these values are not correct for your type of application or you want to create an empty incore file, you have to create an incore file explicitly.

Creating an Incore File Explicitly

You can create an incore File explicitly using the `CALLNAT` statement `ACTION = 'CREATE'`.

The additional parameters are:

Parameter	Meaning
`FILE-IDENTIFIER`	The identifier to be assigned
`VIEW`	To take the field definitions and field headers from a View defined to Predict. If this field is omitted, the file layout is taken from the first `STORE` statement.
`IDB-LOG`	Possible values are: `YES`, `NO` (default). If `IDB-LOG = 'YES'` when creating the Incore-File, it is possible to use `END TRANSACTION/BACKOUT TRANSACTION` (ET/BT) logic. In this case, a log of last changes is kept. A `BACKOUT TRANSACTION` statement reverses all the incore modifications up to the last `END TRANSACTION`. If `IDB-LOG = 'NO', END/BACKOUT TRANSACTION` is ignored (no error produced). If you work without logging, overall performance is better.
`INDEX`	Possible values are: `YES` (default), `NO`. The Incore Database can perform `FIND` operations with or without using an index. Indices are more efficient when large amounts of records are involved, but have fixed overhead. When working without indices, the `*NUMBER` system variable is not available. When performance is important and the file is relatively small, it is recommended that you work without indices. Even with the default option `INDEX=YES`, there are some restrictions regarding support of the system variable `*NUMBER`: ■ With `FIND SORTED`, `*NUMBER` is not available and is set to `9999999`.

Parameter	Meaning
	■ This also applies to complex FIND criteria (FIND ... AND ..., FIND ... OR ...), especially if more than one descriptor is referred to. ■ This also applies to FIND statements which use the comparison operators LT, GT or NE.
ISN-TYPE	Possible values are: STANDARD (default), REUSE, RENUM. The values have the following meaning: STANDARD (Default). An unchanged number is supplied to each new record in an ascending order (same as Adabas). REUSE. Same as STANDARD, but the ISNs of deleted records are reused. RENUM. The ISN is the physical record number. This means that it can change during the record's life. This mode of work is more efficient, and if there is no advantage to be gained from using the *ISN system variable, RENUM is recommended. The ISN can be referred from Natural using the *ISN system variable. The meaning of the ISN depends on the ISN-TYPE of the file. You can also assign a standard ISN to a record. This is a fixed number which does not change throughout the record's life (but note the exception which applies to ISN-TYPE='RENUM', described above.
WILD-CARD-SEARCH	Possible values are: YES, NO (default). Wildcard selection can be supported in search criteria: * (asterisk). Denotes any string of characters. _ (underscore). Placeholder for any character. To enable a wildcard search, WILD-CARD-SEARCH = 'YES' must be specified when creating a new incore file. By default, wildcards are not enabled (WILD-CARD-SEARCH = 'NO').

Example

The following program displays the NAME, AGE, and CITY data for all personnel whose names begin with S and belong to any department that consists of 5 characters and begins with DEV:

```
0270 FIND INCORE-PERSONNEL WITH NAME = 'S*'
0280    AND DEPARTMENT ='DEV__'
0290 DISPLAY NAME AGE CITY
0300 END-FIND
```

The records of an incore file are ordered in a physical sequence. Each record therefore has a physical record number. The record number of a record increases by 1 when a record is inserted before it, or decreases by 1 if a record before it is deleted. In addition to using the ISN, this physical

record number can be referred to directly using the DDM pseudo-field RECORD-NUMBER (Adabas short name RN). The sequence of records may be relevant in cases in which the order of records is significant, for example, TEXT lines.

The following table shows the effect of the ISN-TYPE parameter on ISN assignment as well as the usage of *ISN and RECORD-NUMBER in various situations:

Statement	Value of ISN-TYPE parameter specified for CREATE	Usage of RECORD-NUMBER	ISN assignment / availability
STORE	'STANDARD' (default)	Physical record number can be set in the RN field: if set to nn, the record is inserted and the added record has an RN of *nn*. If RN is zero or not specified, the record is appended to the end of the file.	ISN is automatically generated in ascending order. If this is not intended, use the NUMBER option of the STORE statement to set the ISN explicitly.
	'REUSE'		First unused ISN is allocated. If this is not intended, use the NUMBER option of the STORE statement to set the ISN explicitly.
	'RENUM'		The record is added either at the end of the file or at the position specified in the RN field; in both cases the physical number is the ISN.
READ PHYSICAL		Record number can be returned in the RN pseudo-field.	ISN is returned in the *ISN system variable.
READ BY ISN		RN not available.	ISN is returned in the *ISN system variable.
READ LOGICAL/FIND	'STANDARD' or 'REUSE'	The RN pseudo-field is filled only if the descriptor field referred to in the search criteria is also RN; otherwise it contains zero.	ISN is returned in the *ISN system variable.
	'RENUM'		ISN not available.
GET	'STANDARD' or 'REUSE'	RN not available.	ISN can be used in criteria of GET statement.
	'RENUM'	Record number can be returned in the RN pseudo-field.	ISN can be used in criteria of GET statement.

Creating an Incore File with a WRITE/DISPLAY Statement

You can direct output to an incore file using the WRITE or DISPLAY statement. The file is then created dynamically.

Example

```
0010 DEFINE PRINTER(3) OUTPUT 'INCORE'
0020 FORMAT(3) LS=70
0030 READ(100) PERSONNEL
0040 DISPLAY(3) NAME AGE SEX
0050 END
```

The incore file thus created is assigned the identifier REPORT*nn*, where *nn* is the report number (in the above example, REPORT03). The incore file layout will consist of one field, Type A*xx*, where *xx* is the report line size (in the above example, A70). The Adabas name of the field is A1. The program IDB-REPO in the example library shows how to read an incore file created by a WRITE / DISPLAY statement.

New reports written to INCORE create new incore files. If a new report is written to an incore file with the same identifier, the 'old' file is overwritten. Once an incore file is created using the DISPLAY or WRITE statement, it can be manipulated as described in the subsection *Manipulating Incore Files*.

Manipulating Incore Files with Natural DML

Retrieving Incore File Records

Records can be retrieved using any of the Natural statements READ, FIND, GET and HISTOGRAM.

Example

The following lines display the contents of the incore file identified by DOC:

```
0140 READ TEXT IDENTIFIER = 'DOC'
0150   DISPLAY LINE(AL=70)
0160 END-READ
```

The following lines display all records as specified by the fields from the incore file identified by the default value (L*xxx*, where *xxx* is the DDM file number).

```
0270 FIND INCORE-PERSONNEL WITH NAME = 'SMITH'
0280                       AND AGE  > 27
0290   DISPLAY AGE CITY
0300 END-FIND
```

Note: READ PHYSICAL, READ BY ISN, READ LOGICAL, FIND SORTED, GET *ISN are all supported. The system variables *NUMBER, *ISN are supported with some restriction in some modes of operation. See the explanation of the ISN-TYPE and INDEX parameters in the subsection *Creating an Incore File Explicitly*. If the incore file does not exist, the processing loop returns no records and terminates immediately.

Restrictions

- Binary fields may only be used as secondary search criteria, that is, following AND in a FIND, and may not be used at all for logical reads/histograms.
- In a series of nested FIND or READ LOGICAL processing loops, only 2 may refer to the same file identifier.

Adding Records to an Incore File

The stored record is normally added at the end of existing records. However, you can control the physical sequence of the record in a single operation using the pseudo-field RECORD-NUMBER.

Example

The following program writes ten records with the text "Sample text line no: n" to an incore file, and then inserts the text line this is Line Number 5 to Line 5.

```
0010 DEFINE DATA LOCAL
0020 1 TEXT VIEW OF ISP-IDB-TEXT
0030   2 LINE
0040   2 RECORD-NUMBER
0050 1 I(P3)
0060 END-DEFINE
0070 FOR I = 1 TO 10
0080 COMPRESS 'Sample text line no:'I INTO TEXT.LINE
0090 STORE TEXT
0100 END-FOR
0110 MOVE 'this is Line Number 5'  TO TEXT.LINE
0120 MOVE 5 to TEXT.RECORD-NUMBER
0130 STORE TEXT
```

Note: The view TEXT (a file of lines) is a very useful way of maintaining texts in an incore file. The DDM ISP-IDB-TEXT consists of one field: LINE (A80). This DDM is supplied with the Incore Database facility of Natural ISPF.

Modifying Incore File Records

Records in an incore file can be modified using the Natural statements `UPDATE`, `DELETE`.

Example

This little program updates an incore file by adding 1 to the value of the `AGE` field of a specified record.

```
0270 FIND(1) INCORE-PERSONNEL WITH NAME = 'SMITH'
0280   ADD 1 TO AGE
0290   UPDATE
0300 END-FIND
```

This program deletes all lines containing the string `$TEMP$` from the incore file identified by `DOC1`:

```
0270 READ TEXT IDENTIFIER = 'DOC1'
0280   IF LINE = SCAN '$TEMP$'
0290     DELETE
0300   END-IF
0310 END-READ
```

Managing the Incore Database using the CALLNAT Interface

You can use the `CALLNAT` interface to create, delete, list, edit and browse incore files. Corresponding subprograms are supplied with the Incore Database component of Natural ISPF.

The subprograms are loaded into libraries `SYSISPDB` and `SYSTEM`. If you intend to use Incore Database functionality within your Natural application, make sure that one of the above mentioned libraries is defined as a `STEPLIB` for your application, or copy all modules for `SYSISPDB` into one of your `STEPLIB`s. In addition, you must define library `SYSLIBS` as steplib for your application.

The parameters for the `INCORE` subprogram (`IDBI--NN`) are in a local-data-area: `IDBI---L`, which contains the following three variables:

Parameter	Meaning
`INCORE`	The name of the call subprogram.
`INCORE-CTL`	The first parameter for the subprogram contains control fields for internal use.
`INCORE-DATA`	The second parameter contains all fields described in this section.

The input parameter must be assigned before the `CALLNAT` is issued. The result will be in the data-area fields after the `CALLNAT` execution. Available functions for `CALLNAT` statements to the Incore Database are:

Parameter	Meaning
EDIT	Pass control to the user and allow him to edit incore file contents.
BROWSE	Pass control to the user and display incore file contents.
COMMAND	Issue an Editor command operating on an incore file.
DELETE	Delete an incore file from the Incore Database.
CREATE	**Create an incore file explicitly**.
(blank)	List existing incore files.

Example

These program lines specify the DELETE function and the incore file identifier EX1 before the CALLNAT is issued:

```
....
0630 MOVE 'DELETE' TO ACTION
0640 MOVE 'EX1' TO FILE-IDENTIFIER
0650 CALLNAT INCORE USING INCORE-CTL INCORE-DATA
0660 IF ERROR-CODE > 0
....
```

Deleting an Incore File

You can delete an incore file using a CALLNAT statement with ACTION = 'DELETE' with FILE-IDENTIFIER identifying the incore file to be deleted.

Example

The following lines delete an incore file identified by CURRENT.

```
0810   ASSIGN ACTION = 'DELETE'
0820   ASSIGN FILE-IDENTIFIER = 'CURRENT'
0830   CALLNAT INCORE USING INCORE-CTL INCORE-DATA
....
```

Listing Existing Incore Files

You can list all existing incore files using a CALLNAT statement within a REPEAT loop using ACTION = ' '. Search criteria for the parameter FILE-IDENTIFIER is optional. The FILE-IDENTIFIER and NUMBER-OF-RECORDS are retrieved by the subprogram.

END-OF-DATA contains Y after the last call that returns real data.

Example

The following program displays all incore files whose identifiers begin with R:

```
0900    RESET INCORE-CTL INCORE-DATA
0910    MOVE 'R*' TO INCORE-DATA.FILE-IDENTIFIER
0920    CALLNAT INCORE INCORE-CTL INCORE-DATA
0930    REPEAT UNTIL INCORE-CTL.END-OF-DATA = 'Y'
0940            DISPLAY FILE-IDENTIFIER NUMBER-OF-RECORDS
0950            CALLNAT INCORE INCORE-CTL INCORE-DATA
0960    END-REPEAT
```

Editing/Browsing an Incore File

The display and modification of an incore file on a terminal screen is not as simple as in the case of a record field manipulation. Sometimes, the data will not fit on a single screen page, so that scrolling capabilities are a requirement.

Additionally, editing requires complex functions such as insert characters, delete characters, order text, find/replace string. The incore facility provides such editing capabilities with a single statement.

Using ACTION='EDIT' or ACTION='BROWSE' on the CALLNAT statement, the incore file is presented using the Software AG Editor. The incore processor does all the terminal input output for you. Once the user has finished modifying or viewing the file, control is returned to your Natural program.

Formatted files that contain several fields can also be edited or browsed. In this case, the fields are delimited with a blank. Physical tabulation is available in edit mode. When in edit mode, only fields of the types Alpha and Numeric are supported (Packed and Binary are not supported). If the file contains binary or packed fields, actions EDIT and BROWSE are handled identically.

Example 1

The following lines start an edit session with the incore file identified by MYDOC:

```
....
0510 ASSIGN ACTION = 'EDIT'
0520 ASSIGN FILE-IDENTIFIER='MYDOC'
0530 CALLNAT INCORE USING INCORE-CTL INCORE-DATA
....
```

Incore 1 (data prefixed with 4-digit line number):

```
EDIT:-MYDOC-----------------------------------------------------------------
COMMAND==>
**** ****************************** top of data ***************************
0001 This is the first line of text
0002
0003 It shows an example of an edit session
**** ***************************************************************************









Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12-
      Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

Example 2

The following lines create an incore file dynamically by writing output from the `PERSONNEL` file to it. The first 100 records from the `PERSONNEL` file are written to the incore file, which is assigned the default identifier of `REPORT03`.

```
....
0010 DEFINE PRINTER(3) OUTPUT 'INCORE'
0020 FORMAT(3) LS=70
0030 READ(100) PERSONNEL
0040    DISPLAY(3) NAME CITY SEX
....
```

The following lines browse the newly created incore file `REPORT03` using the `CALLNAT` statement.

```
....
0840  ASSIGN ACTION = 'BROWSE'
0850  ASSIGN FILE-IDENTIFIER = 'REPORT03'
0860  CALLNAT INCORE USING INCORE-CTL INCORE-DATA
....
```

Incore 2 (no prefixes displayed):

```
BROWSE:-REPORT03------------------------------------------------- Row 1 of 149
COMMAND==>
1Page      1                                        93-06-21  09:56:44

         NAME                 CITY         S
                                           E
                                           X
 ------------------- -------------------- -

 GUENTER             JOIGNY               F
 BRAUN               ST-ETIENNE           M
 CAOUDAL             LE BLANC MESNIL      M
 VERDIE              MILLAU               M
 GUERIN              BOULOGNE BILLANCOURT F
 VAUZELLE            MAMERS               M
 CHAPUIS             IVRY SUR SEINE       M
 MONTASSIER          RENNES               M
 JOUSSELIN           PERPIGNAN            M
 BAILLET             LYS LEZ LANNOY       M
 MARX                PARIS                M
 D'AGOSTINO          FONTENAY SOUS BOIS   M
 LEROUGE             ARGENTEUIL           M
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

Additional Parameters when Calling the Editor

When starting a session with the Editor using `ACTION='EDIT'` or `ACTION='BROWSE'`, some additional parameters can be specified to set up the Editor environment:

```
MODIFIED (A3) = ['YES']
                ['NO']
```

This value is returned to the caller when `ACTION='EDIT'` or `ACTION='COMMAND'`.

`YES` means the text has been modified in an edit session.

```
PREFIXES (A6) = ['NUMS']
                ['CMD']
                ['NONE']
```

Default: `NUMS` for `ACTION='EDIT'`; `NONE` for `ACTION='BROWSE'`.

Parameter	Meaning
NUMS	Data is prefixed with a 4-digit line number. For an example, see figure with Example 1.
CMD	Data is prefixed with 2 blanks. For an example, see figure on next page.
NONE	No prefixes are displayed. For an example, see figure with Example 2.

Incore 3 (data prefixed with 2 blanks):

```
  BROWSE:-REPORT01------------------------------------------------- Row 1 of 149
  COMMAND==>
     1Page      1                                      93-06-21  09:36:59

               NAME                 CITY          S
                                                  E
                                                  X
      ------------------- -------------------- -

      GUENTER              JOIGNY               F
      BRAUN                ST-ETIENNE           M
      CAOUDAL              LE BLANC MESNIL      M
      VERDIE               MILLAU               M
      GUERIN               BOULOGNE BILLANCOURT F
      VAUZELLE             MAMERS               M
      CHAPUIS              IVRY SUR SEINE       M
      MONTASSIER           RENNES               M
      JOUSSELIN            PERPIGNAN            M
      BAILLET              LYS LEZ LANNOY       M
      MARX                 PARIS                M
      D'AGOSTINO           FONTENAY SOUS BOIS   M
      LEROUGE              ARGENTEUIL           M
  Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
        Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

```
SHOW-COMMAND-LINE (A3) = ['YES']
                         ['NO']
                         ['USR']
```

Default: YES

Parameter	Meaning
YES	Command line is displayed (COMMAND==>). See figure above.
NO	Command line is not displayed. For an example, see figure below.
USR	Command line is not displayed inside the window, but can be entered into *COM variable in the map displayed by the caller. Commands entered into *COM are executed by IDB routine.

Incore 4 (command line not displayed):

```
BROWSE:-MYDOC------------------------------------------------------ Row 1 of 12




This is a sample text line
```

```
TITLE (A50) = ['text']
              ['NO']
```

Parameter	Meaning
text	Text to be displayed in the top-left corner of the edit screen. For an example, see figure below. Default title is composed of the contents of the fields ACTION and FILE-IDENTIFIER (for example: EDIT-MYDOC); see figure **Incore 1** and figure **Incore 4**.
NO	No default text is generated. Messages only are put into the message line. The position of the message line remains unchanged and is controlled by the user program.

Incore 5 (text displayed in top-left corner of screen):

```
This is a program-supplied title-----------------------------------------------

COMMAND==>

0001 This is a sample text line


**** **************************** bottom of data ****************************
```

```
MESSAGE (A50) = ['text']
```

Parameter	Meaning
text	Text to be displayed in the top-right corner of the edit screen, the first time the edit screen is displayed. For an example, see figure below.

Incore 6 (text displayed in top-right corner of screen).

```
  This is a program supplied title-------------Please modify text or press PF3!!
  COMMAND==>
  0001 This is a sample text line
  **** ***************************** bottom of data ******************************
  Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
        Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

```
COMMAND (A50) = ['command']
```

A valid Editor command, to be executed by the Editor before the edit screen is displayed.

Example

This program starts an edit session with incore file `TXT1`. The Editor command prompt is not shown, the document name is displayed in the top left corner. When the edit screen is displayed, the cursor will be on the first occurrence of string `SUBJECT`:

```
....
0780 ASSIGN ACTION = 'EDIT'
0790 ASSIGN FILE-IDENTIFIER     = 'TXT1
0800 ASSIGN SHOW-COMMAND-LINE   = 'NO'
0810 ASSIGN TITLE =#DOCUMENT-NAME
0820 ASSIGN COMMAND          = 'FIND SUBJECT'
0830 CALLNAT INCORE USING INCORE-CTL INCORE-DATA
....
```

```
ALARM (A3) = ['YES']
             ['NO']
```

Default: `NO`

Parameter	Meaning
`YES`	The Editor sounds an audible signal the first time the edit screen is displayed (useful together with `MESSAGE`).
`NO`	No signal is sounded.

```
SCROLL (A4) = ['CSR']
              ['PAGE']
              ['HALF']
```

Default: CSR

Parameter	Meaning
`CSR`	Scroll by cursor position.
`PAGE`	Scroll by page.
`HALF`	Scroll by half a page.

Other values such as `WORD` are also allowed, for a full list of valid values, see the Software AG Editor documentation.

```
HORIZONTAL-SCROLL (A3) = ['YES']
                         ['NO']
```

Default: `NO`

Parameter	Meaning
`YES`	Allow horizontal scroll (left/right).
`NO`	Disallow horizontal scroll. The message `cols` *`nnn mmm`* is not displayed, either.

```
HEADER (A80) = ['text']
               ['YES']
```

Parameter	Meaning
text	Text to be displayed above the first line of the data. For an example, see figure Incore 1 below.
YES	If CALLNAT ACTION = 'CREATE' was used with specification of view fields, YES generates the text automatically from the header defined in Predict.

If HEADER is empty, the header line will not be displayed. **Incore 1** (text displayed above first line of data):

```
  Top of the Pops - All-time Greats-------------------------------------------------
   COMMAND==>
        Title                          Group                No.
   **** ********************** top of data **********************
   0001 I Can't Dance                  Genesis              034
   0002 Steel Wheels                   Rolling Stones       077
   0003 Private Dancer                 Tina Turner          089
   0004 Goodbye Cream                  Cream                118
   0005 Abbey Road                     Beatles              234
   0006 Joshua Tree                    U2                   255
   0007 Thriller                       Michael Jackson      276
   0008 Born in the USA                Bruce Springsteen    301
   0009 So                             Peter Gabriel        345
   0010 Nightingales and Bombers       Manfr. Mann's Earthb.412
   **** ******************** bottom of data *********************
   Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
         Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

```
START-HIGHLIGHT-CHAR (A1) = ['char']
END-HIGHLIGHT-CHAR (A1)   = ['char']
```

Special characters that start and end highlighting of text in a browse screen. Highlighting must be started and ended on one line. The START/END characters themselves are displayed as blanks. An example of highlighted text is given in the figure below.

Incore 2 (highlighted text):

```
  BROWSE:-MYDOC------------------------------------------------------ Row 1 of 2
  COMMAND==>
  This word is highlighted
  This word is highlighted
```

```
  Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
        Help        Quit        Rfind Rchan Up    Down  Swap  Right Left  Curso
```

Note that to achieve the highlighting in the above example, each occurrence of "word" must be enclosed in the `START/END` characters.

Editing in Windowing Mode

Instead of editing in full screen mode, you can display the edit screen in a window and edit in the window.

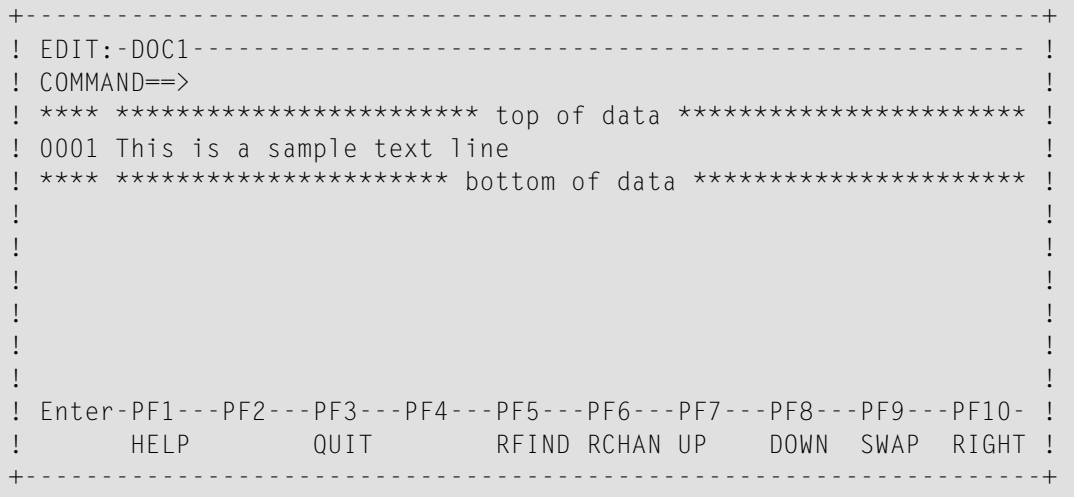
Example

The `DEFINE WINDOW` and `SET WINDOW` statements define the window in which the Editor is displayed:

```
DEFINE WINDOW WIND1 SIZE 15 * 60   '
             BASE 3 / 10
SET WINDOW 'WIND1'
ASSIGN ACTION = 'EDIT'
....
```

Incore 3

```
Enter Incore file Id: DOC1
```



A number of more detailed sample programs are delivered on the Natural ISPF installation medium. For a list of their names, issue the command:

```
HELP EXAMPLE
```

and scroll down to the heading `INCORE DATABASE Examples`.

PF Key Handling

The Editor is sensitive to the PF key status, declared in the program. Key position, key display form, key sensitivity and key name can be declared from the program using the `SET KEY` statement. Keys are translated to Editor commands, using the `DATA` clause in the `SET KEY` command.

Example

```
SET KEY PF7=DATA 'UP 4' NAMED 'Up'
SET KEY PF8=DATA 'DOWN 4' NAMED 'Down'
```

PF Key Display

The display of the PF key lines when editing/browsing is controlled by the field:

```
KEYS-TYPE (A8) = {'OFF'}
                 {'ON'}
                 {'DEFAULT'}
                 {'SCREEN'}
```

Possible values are:

Parameter	Meaning
OFF	No control is given to the user by PF keys
ON	Control is given to the user by PF keys. Keys can be defined by the statement SET KEY = DATA *xx* NAMED *yy*, where: *xx* is the Editor command to be executed when the PF key is pressed. *yy* is the text to be displayed in the PF key lines. The two bottom lines of the screen or window are reserved for display of defined keys. **Note:** PF keys are evaluated by the incore processor and are reset when returning to the caller. For this reason it is recommended that you define the PF keys every time before invoking the incore processor with ACTION = 'EDIT' / ACTION = 'BROWSE'.
DEFAULT	Keys are active, Natural ISPF default PF keys are active regardless of the user keys definitions.
SCREEN	Same as ON, but no lines are reserved for display of PF keys (to be used only with DEFINE WINDOW CONTROL SCREEN clause).

If no key is defined, the Natural ISPF default keys are used.

Assigning Default PF Keys and Language-dependent Constants

A user exit IDB-USRN, distributed in the User Exit Library, can be used to set default PF keys and some language-dependent constants. In order to use this exit you have to copy IDB-USRN to your library and modify it according to your needs.

Escape Commands

After you have edited or browsed an incore file, the QUIT command returns control to the program. However, it is possible to pass escape commands to the Editor. These are commands that, when entered in the edit session, also return control to the program, which then processes the command.

The following parameters can be used on the CALLNAT:

```
ESCAPE-MAIN-COMMANDS (A120) = {'command1 command2 ...'}
{*}
```

Passes escape commands to the Editor. Multiple commands must be separated by a blank. The asterisk (*) as value can be used to pass all commands which cannot be processed by the incore processor to the caller.

```
RETURN- MAIN-COMMAND (A10) = 'token'
```

When control is returned to the program, the first token of the command is returned in this field.

```
RETURN-MAIN-COMMAND-PARM (A10) = 'text'
```

The rest of the command is returned in this field.

Example

The following program starts an edit session with incore file TXT and passes the escape commands CLEAR and ZIP to the Editor.

```
1120 ASSIGN ACTION ='EDIT'
1130 ASSIGN FILE-IDENTIFIER = 'TXT1'
1140 ASSIGN ESCAPE-MAIN-COMMANDS = 'CLEAR ZIP'
1150 CALLNAT INCORE USING INCORE-DATA INCORE-CTL
1160 DECIDE ON FIRST VALUE OF RETURN-MAIN-COMMAND
1170 VALUE ' '     IGNORE
1180 VALUE 'CLEAR' PERFORM DELETE-TEXT
1190 ESCAPE TOP  /* BACK TOP PROCESS EDIT
.....
```

Following the same logic, escape line commands can also be used. The available parameters are:

```
ESCAPE-LINE-COMMANDS (A36) = 'command'
```

Passes the line command(s) to the Editor.

```
RETURN-LINE-COMMAND (A2) = 'command'
```

When an escape line command is entered in the edit session, the program receives the line command in this field.

```
RETURN-LINE-DATA (A80) = 'data'
```

When an escape line command is entered in the edit session, the program receives the text in this line in this field.

```
RETURN-LINE-NUMBER (N7) = 'number'
```

When an escape line command is entered in the edit session, the program receives the relative line number in this field.

Example

```
1250 ASSIGN ACTION = 'EDIT'
1260 ASSIGN FILE-IDENTIFIER = 'TXT1' ,
1270 ASSIGN ESCAPE-LINE-COMMANDS = 'TL'
1280 CALLNAT INCORE USING INCORE-DATA INCORE-CTL
1290 DECIDE ON FIRST VALUE OF RETURN-LINE-COMMAND
1300 VALUE ' '  IGNORE
1310 VALUE 'TL
1320 CALLNAT 'TRANSLAT' RETURN-LINE-DATA
1330 FIND(1) TEXT IDENTIFIER = 'TXT1'
1340 WITH RECORD-NUMBER = RETURN-LINE-NUMBER
1350 MOVE RETURN-LINE-DATA TO LINE
1360 UPDATE
....
1450 END-FIND
```

Resetting the Change Flag

If the contents of an incore file has been saved (stored) in the database and the incore file is not deleted, it could be necessary to mark the incore file as not being modified since the last update of the database. This can be done by using the `RESETMOD` action. No further parameters are required.

Example

The following program lines specify the RESETMOD function:

```
....
0440 MOVE 'RESETMOD' TO ACTION
0450 MOVE 'EX1' TO FILE-IDENTIFIER
0460 CALLNAT INCORE USING INCORE-CTL INCORE-DATA
0470 IF ERROR-CODE > 0
....
```

Editor Commands

All Editor commands are described in detail in the Software AG Editor documentation. The following commands are supported when editing incore files:

Main Commands

Display commands:

```
BNDS,COLS MASK
CAPS,HEX,NULLS,ADVANCE,ESCAPE,EMPTY,FIX,PROTECT
TABCHAR,TABS,LTAB,
EXCLUDE,INCLUDE,XSWAP
PROFILE,RESET
```

Position commands:

```
BOTTOM,TOP,DOWN,UP,LEFT,RIGHT,-H,-P,+H,+P
CURSOR,HOME
LABEL LOCATE,LX,POINT
```

Text commands:

```
CHANGE,FIND,RFIND,RCHANGE
DELETE,DX,DY,CX,CY
SHIFT
LC,UC
RENUMBER,UNREN
POWER,ORDER
CENTER,JLEFT,JRIGHT,JUSTIFY
WINDOW,CWINDOW,MWINDOW,DWINDOW
```

Line Commands:

Positioning:

```
A,B,O,T
```

Display:

```
X,F,L
```

Text handling:

```
I,D,R,C,M,W,N
),(,>,<
S,J,UC,LC
BNDS,COLS,MASK,TABS
TF,TC,LJ,RJ,TI,TE
CX,CY,DX,DY,MX,MY,CX-Y,DX-Y,MX-Y,CY-
X,DY-X,MY-X
WS,WM,WC,WM
```

Issuing Edit Commands using CALLNAT

You can issue an edit command to an incore file using `ACTION = 'COMMAND'` on the `CALLNAT` statement to `INCORE`. With this mechanism program controlled editing can be implemented in an easy way. If the command modified data, `YES` is returned in the `MODIFIED` field.

Example

The following lines issue the `EDITOR CHANGE` command to an incore file identified by `REPORT03`:

```
....
0840    ASSIGN ACTION  = 'COMMAND',
0850    ASSIGN FILE-IDENTIFIER = 'REPORT03'
0860    ASSIGN COMMAND = 'Change ATKIN ADKIN all'
0870    CALLNAT INCORE USING INCORE-CTL INCORE-DATA
....
```

Error Handling

If an error occurs during the execution of the CALLNAT statement, it is returned to the ERROR-CODE and ERROR-TEXT fields.

CALLNAT Parameter Summary

The following matrix provides an overview of which parameters on the CALLNAT statement are relevant to which actions on incore files:

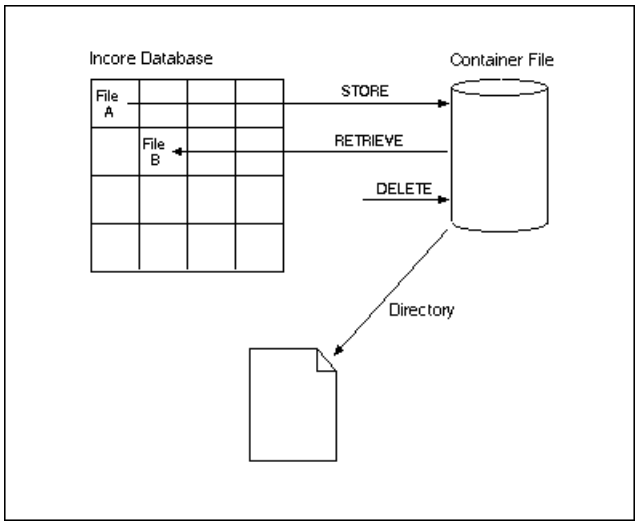
I=Input
O=Output
R=Required

Action / Field	CREATE	EDIT	BROWSE	DELETE	COMMAND	blank
ACTION	IR	IR	IR	IR	IR	IR
FILE-IDENTIFIER	IR	IR	IR	IR	IR	IO
END-OF-DATA						O
ERROR-CODE	O	O	O	O	O	O
ERROR-TEXT	O	O	O	O	O	O
NUMBER-OF-RECORDS						O
PREFIXES		I	I			
SHOW-COMMAND-LINE		I	I			
TITLE		I	I			
ESCAPE-MAIN-COMMANDS		I	I			
ESCAPE-LINE-COMMANDS		I	I			
HORIZONTAL-SCROLL		I	I			
COMMAND		I	I		I	
HEADER		I	I			
MESSAGE		I	I			
ALARM		I	I			
START-HIGHLIGHT-CHAR			I			
END-HIGHLIGHT-CHAR			I			
SCROLL		I	I			
KEYS-TYPE		I	I			
MODIFIED		O			O	
RETURN-LINE-COMMAND		O	O			
RETURN-LINE-DATA		O	O			
RETURN-LINE-NUMBER		O	O			

Action / Field	CREATE	EDIT	BROWSE	DELETE	COMMAND	blank
RETURN-MAIN-COMMAND		O	O			
RETURN-MAIN-COMMAND-PARM		O	O			
IDB-LOG	I					
INDEX	I					
ISN-TYPE	I					
VIEW	I					
WILD-CARD-SEARCH	I					

The Incore Database Container Data Set

You can use a "container" data set in which you can store incore files for later retrieval. The container data set itself can be an Adabas or VSAM file. The following figure illustrates the container data set concept:



You can access this container using a CALLNAT statement in the same way as for INCORE, this subprogram is called IDBC---N.

Local data containing the subprogram parameters are supplied in a local-data-area called IDBC---L. The physical container data set is preset using the LFILE parameter (see the *Natural ISPF Administration Guide*).

END TRANSACTION is not performed by these container accesses, and must be performed by the user program.

Available functions for CALLNAT statements to the container data set are:

Function	Meaning
STORE	Store an incore file into the container data set.
RETRIEVE	Retrieve an incore file from the container data set.
DELETE	Delete an incore file from the container data set.
(blank)	Retrieve a directory of the container data set.

An incore file saved in a container data set is identified by means of three fields:

- TYPE (A8)
- GROUP (A48)
- NAME (A32)

Example

The following lines store the incore file identified by PERS in the container data set. The file can later be identified using the values of the TYPE, GROUP and NAME fields:

```
0510 ASSIGN ACTION          = 'STORE'
0520 ASSIGN FILE-IDENTIFIER = 'PERS'
0530 ASSIGN TYPE            = 'APP1'
0540 ASSIGN GROUP           = 'PERSONS'
0550 ASSIGN NAME            = 'REP001'
0560 CALLNAT CONTAINER USING CONTAINER-CTL CONTAINER-DATA
```

The following lines retrieve this incore file from the container data set, assigning the identifier PERS1:

```
0610 ASSIGN ACTION          = 'RETRIEVE'
0620 ASSIGN FILE-IDENTIFIER = 'PERS1'
0630 ASSIGN TYPE            = 'APP1'
0640 ASSIGN GROUP           = 'PERSONS'
0650 ASSIGN NAME            = 'REP001'
0660 CALLNAT CONTAINER USING CONTAINER-CTL CONTAINER-DATA
```

Lists of files in the container data set can be generated in the same way as listing incore files.

Examples

The following program lists all container entries of type `APP1` and group `PERSONS`:

```
0760 ASSIGN ACTION          = ' '
0770 ASSIGN TYPE            = 'APP1'
0780 ASSIGN GROUP           = 'PERSONS'
0790 CALLNAT CONTAINER USING CONTAINER-CTL CONTAINER-DATA
0800 REPEAT UNTIL END-OF-DATA = 'Y'
0810   DISPLAY TYPE GROUP NAME
0820   CALLNAT CONTAINER USING CONTAINER-CTL CONTAINER-DATA
0830 END-REPEAT
```

The following deletes the incore file identified by the `TYPE`, `GROUP` and `NAME` fields from the container data set:

```
0930 ASSIGN ACTION          = 'DELETE'
0940 ASSIGN TYPE            = 'APP1'
0950 ASSIGN GROUP           = 'PERSONS'
0960 ASSIGN NAME            = 'REP001'
0970 CALLNAT CONTAINER USING CONTAINER-CTL CONTAINER-DATA
```

3 Open NSPF

The Open NSPF facility enables you to modify and enhance Natural ISPF according to the specific needs of your site. This is easily done by writing site-specific logic in user-exits, while keeping all the advantages of the Natural ISPF environment such as split-screen, multi-session and command logic.

Overview

Natural ISPF is an integrated product, that enables you to work with different external objects (such as `PDS`, `NATURAL`, `JOBS`) in a unified environment. Hence, Natural ISPF functionality is provided by means of objects (for example, `PDS MEMBER`) that are accessed by functions (for example, `EDIT`). The unified environment is presented to the user by means of menus (such as the Administrator Menu) and commands (such as `TECH`).

- **Customized Menus:**
  Natural ISPF menus are built from a screen layout and a command related to each option. In Open NSPF, it is possible to define new menus, to alter existing menus and change the default Main Menu. This option has existed in Natural ISPF since Version 1.1 and is documented in the *Natural ISPF Administration Guide*. For example, you can add the option `Predict` to your `Main Menu`.
- **Customized Commands:**
  A Natural ISPF command can be executed from any place in Natural ISPF and is not related to any specific object. Open NSPF enables you to define new commands. These commands are directed at user-written subprograms. You can also use this facility to define command synonyms or to overwrite existing commands. For example, you can define `MAIL` as a new command that checks whether there is mail waiting for you (in Con-nect or another site-specific office system).
- **Customized Objects and Functions:**
  A Natural ISPF object usually references external objects or data which can be read, modified or edited (for example, a library member). Each object can be identified by several fields. For example, the object `PDS` is identified by the fields `DSNAME`, `MEMBER`, `VOLSER` and `NODE`. A Natural ISPF object can be related to one or several (new or existing) functions such as `EDIT`, `BROWSE`. Each activation of a function for an object invokes a Natural ISPF panel, a full screen that can be suspended and resumed according to Natural ISPF logic, until it is terminated by the user, or by the logic of the function (for example, `DELETE`). For example, you can add new object `EMPLOYEE` and relate it to functions `LIST`, `DELETE` and `INFORMATION`.

You can also add a new function and relate it to an existing object. Since Natural ISPF does not know the new function, you have to maintain it in the object user exit and transfer control to another user-defined object, which contains the logic to be executed (see also the subsection *Object Exits* in the section *User Exits* in the *Natural ISPF Administration Guide*).

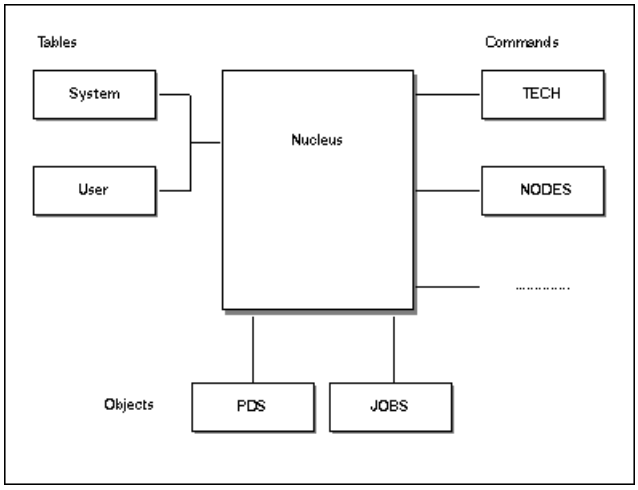
Customizing Natural ISPF

Natural ISPF architecture can be summarized by listing the following modules:

- **Nucleus:**
  The nucleus is responsible for all the common logic, for example, the logic supported by the following main modules:
  - **Command Processor:**
    The Command Processor interprets commands typed in by the user.
  - **Browser:**
    The Browser displays `LIST/BROWSE/EDIT` sessions on the screen.
  - **Manager:**
    The Manager supports multiple sessions and split screen.

  The nucleus is responsible for the screen I/O of the Editor, but not for the object-specific screens. The nucleus is the main part of Natural ISPF. It is written mostly in Natural and executed from `SYSLIB`.
- **Natural ISPF tables:**
  The Natural ISPF tables are stored in the Natural system file and contain definitions such as the existing objects, their names and synonyms, the functions, the commands, menus, user profiles etc. Natural ISPF is installed with predefined tables, stored in the System Profile Library (`SYSISPS1`). Site-specific tables that can extend or overwrite the predefined tables are stored in the User Profile Library (`SYSISPFU`).
- **Command Modules:**
  There is a module for each command, implementing the command logic. The command modules are written as Natural subprograms, called from the Natural ISPF nucleus, and operate from `SYSLIB`.
- **Object Modules:**
  There is one module for each object, responsible for the logic specific to this object. For example, the PDS module implements reading a member from a PDS library, writing it, deleting it, displaying the PDS entry panel, displaying information screen etc. The object modules are written as Natural subprograms, called from the Natural ISPF nucleus, and operate from `SYSLIB`.

This architecture can be illustrated as follows:



Multi-Operations Management

One of the attractive features of Natural ISPF is its multi-session management. A user can work in many sessions simultaneously. For example, he can be editing a PDS member while looking at a `JOB` output. This is done using the Natural ISPF Multi-Operation Environment. It is important to understand this mechanism in order to work properly with Open NSPF.

An operation in Natural ISPF is a series of actions within a given context, that can be interspersed with other operations (from the user point of view, an operation can be suspended and resumed). For example, an operation could be issuing a direct command to edit a PDS member, changing a few lines, saving the member, and ending the session. Other examples of operations in Natural ISPF are:

- Performing a function for an object (for example, `BROWSE JOB`)
- Activating a menu
- Special nucleus operations (such as display of PF key assignments)

An operation in Natural ISPF is built of events, where each event is a piece of code that must be executed as a whole (the operation cannot be suspended in the middle of an event). When an operation starts, a special area called “operation data” is allocated, the data in this area is the only data that stays “alive” between the events of the operation. This data is released when the operation ends. This operation data is the only place where data involving the operation can 'live' outside any event, and this is also the preferred communication area between events.

In Open NSPF, operations are involved when implementing customized objects. Note the following:

- The operation starts when a direct command for a function is issued to this object (either by the user or by an open module), or when a line command is issued from a list of this object type.
- When an operation starts, the operation data is allocated, then a series of events for this object is issued, each event is a `CALLNAT` to the object subprogram, where the operation data is passed as a parameter.

- An operation ends when the command `END` is issued to it. This command can be issued by the user or by an open-module.

  Note: A command is not an operation, as only a single call is made to the command module, and no operation data is needed. Displaying a menu is internally an operation, but since it is handled by the nucleus, this is irrelevant to the Open NSPF programmer.

User Objects and User Commands

Defining new user objects and new user commands is done by adding an object code or a command code to the Site Control Table. User objects can additionally be related to existing Natural ISPF functions.

The Site Control Table resides in the User Profile Library and is usually called `CONTROLU`. It can be accessed for update operations with the command `EDIT CNF CONTROLU`, or from the `CONTROLU` (Edit Site Control Table) option on the Configuration Menu.

Implementing the corresponding logic in Natural ISPF is done by writing a subprogram and copying it to the Natural ISPF Execution Library (`SYSLIB`). Each new object and command has a unique program, which is called by Natural ISPF whenever the object is accessed or the command is issued. The naming convention for the Open NSPF subprograms is as follows:

```
ISUxnn
```

where:

`ISU`	Fixed prefix for Open NSPF.
x	`O` - If the program is for a user object. `C` - If the program is for a user command.
nn	Two-letter code of the routine as defined in the Site Control Table.

Example:

The object `EMPLOYEE` will use the code `EM`, which means a subprogram named `ISUOEM` must handle all logic for the object `EMPLOYEE`.

Common Subjects of Open NSPF Routines

The following subjects are valid for all Open NSPF routines, that is, for routines implementing new objects or new commands. This subsection contains reference information and can be skipped if you are reading this section for the first time to gain an idea of Open NSPF.

Natural ISPF Error Handling

For the mechanism of the error handling, see the description of the following fields:

Field Name	Length	Type
INPUT-ERROR-CODE	(N3)	Input

If not equal to 0, Natural ISPF is in error-mode. Error mode usually means skipping further action and displaying the error (acoustic alarm signal and message text in the appropriate part of the HEADER field) to the user in the next input operation. Error mode is reset automatically after the subsequent input operation.

Field Name	Length	Type
OUTPUT-ERROR-CODE	(N3)	Output

If an error occurs in a user subprogram, this field must be set to a non-zero value. The current function is aborted and the error is displayed on the next screen. Additional information about the error must be supplied in the fields ERROR-NUMBER, ERROR-TEXT and ERROR-PARM. OUTPUT-ERROR-CODE should not be set in the DISPLAY event.

Field Name	Length	Type
ERROR-NUMBER	(N4)	Output

If ERROR-TEXT is blank, this field contains the number of the error to be taken from the Error Message Library. The Natural library if the number is in range 0001 - 6799, library SYSISPS1 if the number is in range 6800 - 9999.

Error numbers 9000 - 9999 are not used by Natural ISPF and can be used for site-specific messages.

Field Name	Length	Type
ERROR-TEXT	(A75)	Output

Contains a text that will be displayed in the next input operation (not necessarily in error mode).

Field Name	Length	Type
ERROR-PARM	(A75)	Output

Contains parameters for the error text separated by a semi-colon (;). Parameters in the error-text are noted as :1: . . :2:. Parameter substitution is done by Natural ISPF.

Command Variable

A command variable is passed to every user subprogram. This variable contains the current requested command, if it has not been processed by Natural ISPF yet. If the subprogram changes this field, the commands in the field are pushed to the Natural ISPF command stack.

Data Usage in an Open NSPF Routine

Several data areas are passed to an Open NSPF routine as parameters:

OPERATION-DATA	Local data for Open NSPF routine. The data is kept between events and lives as long as the current function is active. This can be used to save all data necessary to identify the current object.	
GLOBAL-DATA	Shared by all Open NSPF routines, as well as by user exits, and can be used to communicate between these programs. An example for using GLOBAL-DATA can be found in the Example Library:	
	ISUCPR	Implements PREFIX command and stores GLOBAL-DATA.
	ISPJ---U	Job user exit that evaluates GLOBAL-DATA.
STATIC-DATA	Additional shared data. For details on usage, see below.	

Natural ISPF Static Data Usage

Open NSPF routines are subprograms which cannot use a Natural Global Data Area for data shared in several programs. In case this type of data is needed, Natural ISPF offers a mechanism to create and retrieve “static data” which is accessible by all Open NSPF routines via the Natural ISPF data manager.

Natural ISPF can store and retrieve data items throughout the session. The items have a length of 253 bytes and are identified by two letters. They are passed to every Open NSPF routine in two parameters:

```
STATIC-DATA (A253)
STATIC-ID (A2)
```

By default, the data item identified by a blank is passed first, and the data item which was last used is passed. You can modify `STATIC-DATA`. If in a call to the Open NSPF routine the `STATIC-ID` is changed, Natural ISPF will call again with the same event and will pass the static data item that was requested. This is possible for user objects and user commands.

A coding example in an Open NSPF subroutine:

```
DEFINE DATA PARAMETER
  USING ISP-UO-A
PARAMETER
1 #STATIC-DATA(A253)
1 REDEFINE #STATIC-DATA     /* user redefinition of STATIC-DATA
  2 #MY-FIELD1  (A10)
  2 #MY-FIELD2  (N05)
  2 #MY-FIELD3  (A32)
1 #GLOBAL-DATA(A32)
1 #OPERATION-DATA(A253)
1 REDEFINE #OPERATION-DATA
  .....
  .....
END-DEFINE
*
IF #STATIC-ID NE 'PP'          /* get static data with id=PP
  MOVE 'PP' TO #STATIC-ID
  ESCAPE ROUTINE               /* return to get data
END-IF
IF #STATIC-DATA EQ ' '         /* empty just created
  RESET  #MY-FIELD1
         #MY-FIELD2
         #MY-FIELD3            /* set initial values
END-IF
  .....                        /* now it can be used
  .....
```

Defining a User Object

Open NSPF routines to implement new user objects are of type Natural subprogram with a pre-defined parameter area to communicate with Natural ISPF:

```
DEFINE DATA
PARAMETER USING ISP-UO-A      /* Standard Open NSPF interface
PARAMETER
1 #STATIC-DATA        (A253)
1 #GLOBAL-DATA        (A32)   /* Shared data for Open NSPF routine
1 #OPERATION-DATA     (A253)  /* Local data for Open NSPF routine
LOCAL ......
END-DEFINE
```

The parameter area `ISP-UO-A` can be found in the User Exit Library. The Open NSPF routine is called from Natural ISPF every time object-specific logic is to be executed, that is, when the object is accessed by a related function.

The object-specific logic identified by the `EVENT` field in the parameter area is referred to as an event (see the subsection ***Event Logic***).

To add a new object to Natural ISPF, proceed as follows:

1. Allocate a two-letter code to the object (to determine the subprogram name). Object codes should start with an alpha character, special characters and numbers are reserved for Software AG.
2. Prepare a Natural subprogram to handle the object and copy it to `SYSLIB`.
3. Add the object to the Site Control Table.
4. Relate the object to functions. It is recommended that you use existing Natural ISPF functions, but you can also define new functions.

Once the object is defined to Natural ISPF, the object program can be invoked using the Natural ISPF command

```
FF OO parameters
```

where:

`FF`	is the function ID.
`OO`	is the object ID (object abbreviation).

This results in a call to a program with the name `ISUO`*nn*, where *nn* stands for the two-character code identifying the object.

Note: One of the related functions could also be the function `ENTRY`, which presents an Entry Panel, a screen which allows field-oriented input of all parameters relevant for the object (typically the components of `OPERATION-DATA`). The command `ENTRY OO` can then be inserted in one of your site-specific menu definitions, thus making it available within your site-specific menu structure (see the explanation for `ENTRY` in the section *Menu Maintenance* of the *Natural ISPF Administration Guide*).

Site Control Table: Adding a User Object

The Site Control Table can be found in the User Profile Library and is usually called CONTROLU. In this table, you can define new objects, and you can relate objects to functions.

Edit macro MAC-CNFZ is available when editing the Site Control Table. If you wish to use this edit macro, you must use the Natural utility SYSMAIN to copy the following programs from the Example Library (SYSISPE) to the User Profile Library (SYSISPFU):

```
MAC-CNF*
MACCNF*
```

Note: As an alternative, it would also be sufficient to define the library SYSISPE as a STEPLIB for the library SYSISPFU.

- To create a new CONTROLU member, you can use the edit macro with the function command:

```
EDIT CNF CONTROLU MODEL=MAC-CNFZ
```

- To modify an existing CONTROLU member, use the command:

```
REGENERATE
```

in the edit session with the existing member.

The following is an example of a Site Control Table:

```
* OBJECTS
*
*CODE
*  !SUB-system
*  ! !Object sec
*  ! ! !1 letter abbv
*  ! ! ! !3 letter abbv
*  ! ! ! !   !name     !description     !type
*  ! ! ! !   !         !                !
>UU!I! ! !PRU!PRUSERS  !PROCESS users   !U
>E7! ! ! !EMP!EMPLOYEES!Employees       !U
>-9! ! ! !TXT!Text     !Text Members    !U
```

Note: The column delimiting character (!) used in the above example is keyboard-language dependent and corresponds to hex code 4F.

Parameter	Meaning
CODE	Two-character code to be used for the subprogram name. It is strongly recommended that you use a letter as first character. For example: E7. This means that the subprogram name must be ISUOE7.
SUB-System	One-character code of the subsystem to which the object applies. The subsystem codes are the same as used in the Configuration Table, for example, M for z/OS (MVS). If the subsystem is not installed, the object is not available. If no subsystem is specified, the object is always available. For a list of available subsystems, see *Subsystems Supported by Natural ISPF* of the *Natural ISPF Administration Guide*.
Object sec	Authorization class (see ***Authorization Classes*** in this documentation). If you want to restrict access to this object/function it is recommended that you use the '=' (USER DEFINED) authorization class and assign different authorization levels to user/user groups.
1Letter abbv	An object can be abbreviated with 1 letter (as N for Natural), but you *should not* use this 1-byte abbreviation because most of them are already used by Software AG.
3Letter abbv	A 2- or 3-character ID to abbreviate the object in function commands, for example, EMP for EMPLOYEES.
Name	Full name of the object, for example, EMPLOYEES.
Description	Further description of the object used in active help screens.
Type	Identification of a user-defined object. This column *must* contain the letter U for every definition of a user-defined object.

Site Control Table: Adding a User Function

You can define new functions in the Site Control Table CONTROLU in lines starting with the minus sign (-), for example:

```
* FUNCTIONS
*Code
*  !1 Letter abv
*  ! !2 Letter abv
*  ! !  !Name
*  ! !               !Action
*  ! !                        !Security
*  ! !                        !!Parameters?
*  ! !                        !!!Prompt-type
*  ! !                        !!!!Editor?
*  ! !                        !!!! !
-IS! !IS!INSPECT     !*Insp'td!1   !
-RM! !RM!REMARK      !*Remarkd!2  X!
```

Note: The column delimiting character ! used in the above example is keyboard-language dependent and corresponds to hex code 4F.

Parameter	Meaning
CODE	Two-character identifier of function (passed to the subprogram), for example, RM for the function REMARK.
1Letter abbv	A function can be abbreviated with 1 letter (as E for Edit), but you *should not* use this 1-byte abbreviation because most of them are already used by Software AG.
2Letter abbv	Two-letter abbreviation of the function (to be used as line command).
Name	Full name of the function.
Action	Associated attribute text, to be used as reply to line commands in list sessions; max. 8 characters, first character should be an asterisk (*).
Security	Security level assigned, to be compared with the user's authorization level (a digit in the range 1-9). The function can be activated only if the user has an authorization level greater or equal to the security level assigned to the function.
Parameters	Leave blank - for future use.
Prompt-type	Leave blank - for future use.
Editor	Specify X to indicate that function invokes an Editor session (session-type E/B/L/R). Leave blank to indicate that the function results in a message or in a screen handled by the object's user subprogram.

Site Control Table: Relating User Objects to Functions

You can relate the new object to Natural ISPF functions in the Site Control Table as follows:

```
* FUNCTIONS FOR OBJECTS
*
*
*CODE
*  !FUNCTION-OPTION  OPTIONS = D - Default function  X - regular
*  !    !    !    !    !    !    !    !    !
$E7!LS-D!---X!IN-X!DL-X!    !    !    !    !
$-9!LS-D!BR-X!ED-X!DL-X!---X!    !    !    !
$UU!LS-D!---X!DL-X!    !    !    !    !    !
```

Note: The column delimiting character ! used in the above example is keyboard-language dependent and corresponds to hex code 4F.

Parameter	Meaning	
CODE	Two-character code of the object.	
FUNCTION-OPTIONS	In each of these columns, you can define a function that can be applied to the object. A maximum of 10 functions can be activated per object. Each function definition consists of 4 bytes: AABC, where:	
	AA	Function code, for all available function codes see the section *User Exits* in the *Natural ISPF Administration Guide*. Two hyphens (--) as function code means ENTRY function, that is displaying an Entry Panel related to this object.

Parameter	Meaning	
	B	Not used.
	C	Function type: X is a regular function, D the default function. For example, for object E7 (EMPLOYEES): LS-D means LIST is the default, ---X means Entry Panel for the object, IN-X means INFORMATION as a non-default function, DL-X means DELETE as a non-default function.

Example

In the list of active jobs, you want to abbreviate the line command MODIFY with MO, which prompts for an operator command to be sent to the selected active job. Standard Natural ISPF does not support this functionality, but with Open NSPF it could be implemented as follows.

The following definitions must be specified in the Site Control Table:

```
*
* Define a new function MODIFY
*
-MO! !MO!MODIFY      !*Modifd !1   !
*
* Define a new object TASK
*
>-A! ! ! !TAS!TASK     !Tasks           !U
*
* Relate the new function to active jobs (code A)
* and to new object TASK
*
$A !MO-X!    !    !    !    !    !    !    !
$-A!MO-D!    !    !    !    !    !    !    !
```

Now you must implement ISUO-A which contains the logic for the MODIFY command (a coding example can be found in the Example Library). When an MO line command is entered in the list of active jobs you must supply logic to invoke ISUO-A. This can be done with an object transfer in the active jobs user exit ISPA---U (a coding example can be found in the Example Library).

Event Logic

Events are passed to the user-subprogram in the EVENT field of the parameter area. The subprogram must be able to react to the event by executing some object-specific logic for all functions defined for this object. Of course, the subprogram can use other routines to handle an event.

Unknown events must be ignored by the subprogram to allow for the addition of events in later versions of Natural ISPF.

Session Types

Natural ISPF can handle different types of sessions for Natural ISPF objects, as well as for user objects:

Type	Description
' '	The session is handled by the user subprogram. Usually, performing a function means entering data and reacting to user commands, but this is not always the case. For example, a function such as DELETE can operate without additional terminal I/O and then terminate. The user subprogram must: ■ Perform the function, this includes handling the terminal I/O (event=PERFORM). ■ Redisplay the last screen, if terminal I/O has been performed (event=DISPLAY).
E	The session is an Editor session, all terminal I/O is handled by Natural ISPF. All editing commands are allowed in this session. The user subprogram must: ■ Retrieve the data to be edited and store it in an incore file (event=START). ■ Retrieve the data from the incore file and store it in an appropriate place, when a command such as SAVE has been entered (event=COMMAND).
B	A Browse session is very similar to an edit session, the only difference is that update commands are not allowed. In this case, the subprogram does not have to be prepared to save the data.
L	The session is an Editor session, which contains a list of items, such as a list of members in a library. All terminal I/O is handled by Natural ISPF. The list can be manipulated with Editor commands, updates are not allowed. Additionally, all function commands defined for the object can be used as line commands. The user subprogram must: ■ Retrieve the data to be listed and store it in an incore file (event=START). ■ React to line commands entered in the list (event=LINE).
R	The session is like a list, but the list is refreshed whenever ENTER is pressed (can be used for displaying data which changes very frequently like the list of active jobs in Natural ISPF). Line command handling is identical to a list session. The user subprogram must: ■ Check parameters and create an incore file (event=START). ■ Delete the old contents of the incore file and read the actual data to be listed and store it in an incore file (event=REFRESH).

Type	Description
	■ React to line commands entered in the list (event=LINE).

Session Types and Events

This table gives an overview which events receive control depending on the session type, and the numbers indicate the normal sequence of events.

	LINE	PARM	PARM-END	START	TITLE	REFRESH	COMMAND	END	PERFORM	DISPLAY	LISTITEM
LIST	1	1		2	3						x
LIST REFRESHABLE	1	1		2	3	4					x
EDIT	1	1		2	3						
BROWSE	1	1		2	3						
SELF-HANDLED	1	1		2	3				4		

Event Description

This subsection provides a detailed description of all events.

LINE

This event is called as first event when the function is invoked with a line command. In the LINE event, the parameters for the current function must be extracted as in the PARM event for direct commands. Therefore, the parameters supplied in LINE-DATA must be separated and written to OPERATION-DATA as in the PARM event. Remember that when designing a list, all identifiers necessary for line command processing should be in the first 100 byte of a line, because this part of a line is passed in the field LINE-DATA. Care must be taken if left/right shifting commands are possible for the Editor session, because the data visible to the user are always delivered by the LINE-DATA field.

PARM

Natural ISPF function commands can be issued with positional and/or keyword parameters. Keyword parameters are recognized as a pair of tokens, separated by the equal sign (=). This event implements parameter passing, and is processed only if parameters are passed. Each parameter is passed in a separate event in the PARM-KEYWORD and PARM-VALUE fields, so successive calls of this event depend on the number of parameters typed in by the user.

- PARM-KEYWORD contains a keyword if the parameter has been typed in as a keyword parameter, or the position if the parameter was entered as a positional parameter.
- PARM-VALUE contains the parameter value. Valid parameters should be stored in OPERATION-DATA for further processing of the function.

Example:

Assume the user issued the command:

```
EDIT MYPROG T NODE=148 VOLSER=DISK01
```

This command results in the following PARM events:

Number	PARM-KEYWORD	PARM-VALUE
1	1	MYPROG
2	2	T
3	NODE	148
4	VOLSER	DISK01

PARM-END

For future use.

START

This event is called after all parameters have been passed with the PARM or LINE event. This event is also executed if no parameters are passed.

Normally, the parameters collected in OPERATION-DATA are checked if they are all available and correct to execute the function. The function can be aborted by setting the field and return to the caller.

In this event, the Session Type (E/B/L/R) must be set. The next screen is displayed either in PERFORM event in the Open NSPF routine or by the Natural ISPF control logic if the Editor is used, depending on the SESSION-TYPE. For the Session Type Edit/Browse/List/Refreshable list (abbreviated respectively as E/B/L/R), an incore file must be created. Except for type R, the file must be filled with data. For type R, the file is filled with data in the REFRESH event.

TITLE

This event is called once after the START event to get the session title. The given title is then available in the TITLE field in successive PERFORM events, or is displayed in the top left corner of the Browser screen.

REFRESH

This event is called for Session Type R before the screen is displayed (the screen is displayed outside the Open NSPF routine). In this event, the contents of the incore file should be refreshed, which usually means delete and fill again with refreshed data.

`COMMAND`

When the session is handled by Natural ISPF (Session Type `E/B/L/R`), a command is routed to the Open NSPF routine when it is not a valid Editor command. When the screen is self-handled (Session Type `' '`), all commands are first routed to this event. The command must be filtered if it is a valid local command for the current function. Commands which are not handled locally must be returned to Natural ISPF. If line commands and main commands are entered simultaneously, the event `LINE` for the new function is executed before the `COMMAND` event.

`END`

This event is called as the last event before session terminates. If an incore file has been created (Session Type `E/B/L/R`), it must be deleted.

`PERFORM`

This event is called when the screen is handled by the Open NSPF routine itself (Session-Type `' '`). Normally an `INPUT WITH TEXT #TITLE` is coded here.

`DISPLAY`

This event is called when the screen handled by the Open NSPF routine (Session-Type `' '`) must be refreshed, for example when an `UNZOOM` command is entered, that is, the current screen should be displayed (`INPUT` statement) and control should be given to Natural ISPF (`ESCAPE` statement), which will handle non-conversational mode.

`LISTITEM`

This event is called when the user enters the new command `ALL` in a `LIST` session. In the `LISTITEM` event, the parameters for the current function must be extracted similarly to the `LINE` event for line commands. Therefore, the identifier of a single object in the list supplied in the field `LINE-DATA` must be extracted and written to the field `ITEM-NAME`. Remember that when designing a list, all identifiers necessary for line command processing should be in the first 100 bytes of a line, because this part of a line is passed in the field `LINE-DATA`. Care must be taken if left/right shifting commands are possible for the incore file, because the data visible to the user are always delivered by the `LINE-DATA` field.

Parameter Description

This subsection provides a detailed description of all parameters passed to and from the Open NSPF routine.

Parameter Name	Length	Type
COMMAND	(A50)	Input/Output

The first token entered in the command line. If a PF key is pressed, the value assigned to the PF key is delivered as command. If a command is entered and a PF key is pressed simultaneously, the contents of the PF key is concatenated before the command. The value returned in the command field will be processed by Natural ISPF. This takes effect in the START, PERFORM, COMMAND and END events and results in invocation of the corresponding function.

Parameter Name	Length	Type
CHANGED	(L)	Input/Output

This flag is set in an Editor session (session type E) if data are modified. It indicates whether the session was changed by the user and therefore an update must be done. This is relevant to the COMMAND event when a SAVE command is executed and in the END event where the session is closed.

The flag can also be reset by the subprogram (for example, after a successful SAVE).

Parameter Name	Length	Type
ERROR-NUMBER	(N4)	Input/Output

As input parameter, a non-zero ERROR-NUMBER indicates that a message has to be displayed to the user. The text of the message has already been prepared in the TITLE field.

As output parameter, a non-zero ERROR-NUMBER indicates that the text stored in SYSERR for this number has to be displayed to the user in the next Natural ISPF screen (this could be in an Open NSPF subprogram or in Natural ISPF itself).

See also the field OUTPUT-ERROR-CODE. If OUTPUT-ERROR-CODE is not set (value is zero), information can be passed to the user since the current function is not aborted. The error text is taken according to number ranges from the following libraries:

```
6800 - 8999: SYSISPS1
9000 - 9999: are reserved for the user in SYSISPS1
```

Parameter Name	Length	Type
ERROR-TEXT	(A75)	Output

Overrides ERROR-NUMBER.

Parameter Name	Length	Type
ERROR-PARM	(A75)	Output

The ERROR-PARM tokens delimited by a semicolon (;). Parameters to be substituted in the error texts are denoted as :1: :2:

Parameter Name	Length	Type
FUNCTION	(A2)	Input

The function code as defined in the member CONTROL*x*.

Parameter Name	Length	Type
GLOBAL-DATA	(A32)	Input/Output

Data Area common to all Open NSPF routines.

Parameter Name	Length	Type
HEADER	(A79)	Output

If the Editor is used (Session Type E/B/L/R) the column headers are delivered to the caller in this field. If omitted, no column headers are presented in the Editor session.

Parameter Name	Length	Type
IDENTIFIER	(A8)	Input

Unique identifier created for this session. Can be used as file identifier to the Incore Database. This identifier is available in the START event and all subsequent events.

Parameter Name	Length	Type
INPUT-ERROR-CODE	(N3)	Input

Denotes that there is an error situation, that is, the field OUTPUT-ERROR-CODE was set in a previous function or in Natural ISPF itself. In terms of Natural ISPF, this means that the screen must be presented with the ALARM feature.

Parameter Name	Length	Type
LINE-DATA	(A100)	Input

Contains the Editor line as displayed currently in the screen area. Care must be taken if shift left/right is used. In this case, the visible data on the screen is always delivered in LINE-DATA.

Parameter Name	Length	Type
EVENT	(A8)	Input

Defines the event that is to be handled by the Open NSPF routine. For description and possible values, see the subsection ***Event Description***.

Parameter Name	Length	Type
OUTPUT-ERROR-CODE	(N3)	Output

A non-zero value denotes an error situation to Natural ISPF, that is, the current function is aborted and the error denoted by the fields ERROR-NUMBER, ERROR-TEXT and ERROR-PARM is reported in the previous screen. This should be used in real error situations. If the screen is handled by an Open NSPF routine, the message is brought in the field TITLE and is available in the PERFORM and DISPLAY events. OUTPUT-ERROR-CODE should not be set in the DISPLAY event.

Parameter Name	Length	Type
PARM-KEYWORD	(A50)	Input

Contains a keyword, if the notation KEYWORD=PARM-VALUE was used, or a one-digit number, if the parameter is positional.

Parameter Name	Length	Type
PARM-VALUE	(A50)	Input

Contains the parameter value.

Parameter Name	Length	Type
PF-KEY	(A4)	Output

In the PERFORM event, the PF key pressed must be returned to Natural ISPF so that it can be handled by Natural ISPF. That is, *PF-KEY must be moved to PF-KEY.

Parameter Name	Length	Type
OPERATION-DATA	(A253)	Input/Output

Local data for Open NSPF routine. The data is kept between events and lives as long as the current operation is active.

Parameter Name	Length	Type
SESSION-TYPE	(A2)	Output

Possible values:

' '	The screen is handled by the Open NSPF routine itself in the PERFORM event.
'E'	Editor EDIT mode
'B'	Editor BROWSE mode. No line commands are valid in this type of session.
'L'	Editor LIST mode. All function commands can be entered as line commands in this type of session.
'R'	Editor refreshable LIST mode. All function commands can be entered as line commands in this type of session.

Parameter Name	Length	Type
STATIC-DATA	(A253)	Input/Output

Shared data segment identified by STATIC-ID. The data is always updated when it is changed upon return to Natural ISPF. The data segment lives as long as the Natural ISPF session lives. If the STATIC-DATA and STATIC-ID are changed in one operation, the data is updated for the old ID and the new Segment for the new ID is returned.

Parameter Name	Length	Type
STATIC-ID	(A2)	Input/Output

Identification for a shared data segment. If this ID is changed, the current event is triggered again and the appropriate data segment is returned. The last STATIC-ID accessed is always returned as a default for new functions.

Parameter Name	Length	Type
TITLE	(A79)	Input/Output

The session title which is displayed in the first line of the screen. It is assigned in the event TITLE and used in the events PERFORM and DISPLAY when the screen is handled by the Open NSPF routine itself (Session Type ' '). An error message or error text is brought in the right part of the TITLE when it is requested. This means the fields ERROR-NUMBER and ERROR-TEXT and ERROR-PARM are converted and assigned to the TITLE.

Parameter Name	Length	Type
ITEM-NAME	(A70)	Output

Contains valid parameters which must be extracted from the LINE-DATA field in the LISTITEM event. ITEM-NAME can contain any combination of positional and keyword parameters according to the parameter syntax for the current object. The contents of ITEM-NAME are used by Natural ISPF to generate a function command with parameters as returned in ITEM-NAME. Thus a later PARM event must also be able to interpret these parameters. A coding example can be found in the member ISUO-7 in the Example Library.

The following table gives an overview which parameters take effect depending on the event:

Event / Parameter	PARM	PARM-END	START	TITLE	REFRESH	LINE	COMMAND	END	PERFORM	DISPLAY	LISTITEM
COMMAND			O				IO		O		
CHANGED							I	I			
ERRORNUMBER	O	O	O		O		O	O	O		
ERROR-TEXT	O	O	O		O		O	O	O		
ERROR-PARM	O	O	O		O		O	O	O		
FUNCTION	I	I	I	I	I	I	I	I	I	I	I
GLOBAL-DATA	IO	IO	IO	IO	IO	IO	IO	IO	IO	IO	IO
HEADER			O								
INPUT-ERROR-CODE									I	I	
IDENTIFIER			I		I		I	I	I	I	
LINE-DATA						I					I
EVENT	I	I	I	I	I	I	I	I	I	I	I
OUTPUT-ERROR-CODE	O	O	O		O		O	O	O		
PARM-KEYWORD	I										
PARM-VALUE	I										
PF-KEY									O		
OPERATION-DATA	IO	IO	IO	IO	IO	IO	IO	IO	IO	IO	
SESSION-TYPE			O								
STATIC-DATA	IO	IO	IO	IO		IO	IO	IO	IO	IO	IO
STATIC-ID	IO	IO	IO	IO		IO	IO	IO	IO	IO	IO
TITLE				O					I	I	
ITEM-NAME											O

Note: The PARM-END event is for future use. As new functionality is implemented in future, more events may be created. In order to be upwards compatible with future versions of

Natural ISPF, it is therefore good coding practice if your subprograms ignore unknown or unused events.

Example

The following example can be found in the Example Library:

```
************************************************************************
* OBJECT : ISUO-7  DATE CREATED: 16.02.93      BY: JWO
* ---------------------------------------------------------------------
* PURPOSE:
*   Example program which uses Incore Database(IDB)
*   and OPEN NSPF. The functions list of employees
*   and info employees are implemented and have an
*   NSPF like user-interface
************************************************************************
*
DEFINE DATA PARAMETER
  USING ISP-UO-A
PARAMETER
1 #STATIC-DATA(A253)
1 #GLOBAL-DATA(A32)
1 #OPERATION-DATA(A253)
1 REDEFINE #OPERATION-DATA              /* our memory
  2 #PERSONNEL-ID (A8)
  2 #NAME        (A20)
  2 #FIRST-NAME  (A20)
1 #LINE-DATA(A100)                    /* list line passed when
1 REDEFINE #LINE-DATA                 /* line commands are entered
  2 #LINE-PERSONNEL-ID (A8)
  2 #F1               (A01)
  2 #LINE-FIRST-NAME  (A20)
  2 #F2               (A01)
  2 #LINE-NAME        (A20)
*
LOCAL USING IDBI---L                  /* for Incore database
LOCAL
1 EMPLOYEES  VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SEX
  2 BIRTH
  2 DEPT
  2 JOB-TITLE
1 EMPL-LIST  VIEW OF ISP-IDB-EMPL-LIST /* Incore file to be listed
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 BIRTH
```

```
  2 JOB-TITLE
1 #HEADER2
  2 PERSONNEL-ID  (A8) INIT <'Number'>
  2 #F1           (A1)
  2 FIRST-NAME    (A20) INIT <'First-Name'>
  2 #F2           (A1)
  2 NAME          (A20) INIT <'Name'>
  2 #F3           (A1)
  2 BIRTH         (A6)  INIT <'Birth'>
  2 #F4           (A1)
  2 JOB-TITLE     (A20) INIT <'Title'>
1 REDEFINE #HEADER2
  2 #HEADER1 (A77)
1 #NO-RECORDS(L) INIT <TRUE>
1 #END-NAME (A8)
END-DEFINE
*
* Mainline
* Functions for EMPLOYEES are LIST, INFO and ENTRY PANEL
*
DECIDE ON FIRST VALUE OF #FUNCTION
  VALUE 'LS'
    PERFORM EMPL-LIST
  VALUE 'IN'
    PERFORM EMPL-INFO
  VALUE '--'
    PERFORM EMPL-ENTRY-PANEL
  NONE IGNORE
END-DECIDE
*
*
* Function Subroutines
*
*
DEFINE SUBROUTINE EMPL-LIST
*
DECIDE ON FIRST VALUE OF #EVENT
  VALUE 'LISTITEM'                     /* For ALL command
    PERFORM ITEM-OPTION
*
  VALUE 'PARM'                         /* Get parameters
    PERFORM PARM-OPTION
*
  VALUE 'START'
    IF #NAME = ' '                     /* parameter missing
      MOVE 1 TO #OUTPUT-ERROR-CODE     /* error
      MOVE 6802 TO #ERROR-NUMBER
      ESCAPE ROUTINE
    END-IF
*
*       Fill Incore File (Edit session) with data
*
```

```
    EXAMINE #NAME FOR '*' REPLACE    '  '
    COMPRESS #NAME H'FF' INTO #END-NAME LEAVING NO
*
    READ (100) EMPLOYEES  BY NAME STARTING FROM #NAME
      IF EMPLOYEES.NAME GT #END-NAME
        ESCAPE BOTTOM
      END-IF
      MOVE FALSE TO #NO-RECORDS
      MOVE BY NAME EMPLOYEES TO EMPL-LIST
      STORE EMPL-LIST IDENTIFIER = #IDENTIFIER
    END-READ
*
    IF #NO-RECORDS
      MOVE 1 TO #OUTPUT-ERROR-CODE
      MOVE 'No employee found' TO #ERROR-TEXT
    END-IF
    ASSIGN #SESSION-TYPE = 'L'         /* it is a LIST session
    ASSIGN #HEADER = #HEADER1          /* with field headers
    END TRANSACTION
*
  VALUE 'TITLE'                        /* Create a Title
    COMPRESS #TITLE  #NAME INTO #TITLE
*
  VALUE 'END'                          /* Delete Incore file
    MOVE #IDENTIFIER TO FILE-IDENTIFIER
    MOVE 'DELETE' TO ACTION
    CALLNAT INCORE USING INCORE-CTL INCORE-DATA
*
*
  VALUE 'COMMAND' IGNORE               /* Local command handling
  VALUE 'PARM-END' IGNORE              /* End of parameter parsing
  NONE IGNORE                          /* other events ignored
END-DECIDE
*
END-SUBROUTINE
*
DEFINE SUBROUTINE EMPL-INFO
DECIDE ON FIRST VALUE OF #EVENT
*
  VALUE 'LINE'                         /* Get parameters from list-line
    PERFORM LINE-OPTION
*
  VALUE 'PARM'                         /* Get parameters
    PERFORM PARM-OPTION
*
  VALUE 'START'
    IF #PERSONNEL-ID = ' '             /* Missing parameters
      MOVE 1 TO #OUTPUT-ERROR-CODE
      MOVE 6802 TO #ERROR-NUMBER
      ESCAPE ROUTINE
    END-IF
    ASSIGN #SESSION-TYPE = ' '         /* session is handled in here
```

```
*
  VALUE 'TITLE'                          /* Create a Title
    IF #NAME NE ' '
      COMPRESS #TITLE  #NAME INTO #TITLE
    ELSE
      COMPRESS #TITLE  #PERSONNEL-ID INTO #TITLE
    END-IF
*
  VALUE 'PERFORM' ,'DISPLAY'             /* handle session
    MOVE TRUE  TO #NO-RECORDS
    FIND EMPLOYEES WITH PERSONNEL-ID = #PERSONNEL-ID
      INPUT WITH TEXT #TITLE USING MAP 'ISUO-7IM'
      IF #EVENT = 'DISPLAY'
        ESCAPE ROUTINE
      END-IF
      MOVE *PF-KEY TO #PF-KEY            /* return pressed key for
                                         /* interpretation
      MOVE FALSE TO #NO-RECORDS
    END-FIND
    IF #NO-RECORDS
      MOVE 1 TO #OUTPUT-ERROR-CODE
      MOVE 'No employee found' TO #ERROR-TEXT
      MOVE 'END' TO #COMMAND
    END-IF
  VALUE 'COMMAND' IGNORE                 /* Local command handling
  VALUE 'PARM-END' IGNORE                /* End of parameter parsing
  NONE IGNORE
END-DECIDE
END-SUBROUTINE
*
DEFINE SUBROUTINE EMPL-ENTRY-PANEL
*
DECIDE ON FIRST VALUE OF #EVENT
*
  VALUE 'LINE'                           /* Get parameters from line
    PERFORM LINE-OPTION
*
  VALUE 'PARM'                           /* Get parameters
    PERFORM PARM-OPTION
*
  VALUE 'TITLE'                          /* Create a Title
    MOVE 'EMPLOYEES - ENTRY PANEL' TO #TITLE
*
  VALUE 'PERFORM' ,'DISPLAY'             /* Non Editor functions
    INPUT (AD=MI) WITH TEXT #TITLE
      'COMMAND ===>'(I)  #COMMAND
      /
      /
      /  '   Name         '(I)  '===>'(D) #NAME
      /  '   Personnel-No '(I)  '===>'(D) #PERSONNEL-ID
    IF #EVENT = 'DISPLAY'   ESCAPE ROUTINE END-IF
    IF #COMMAND = ' ' AND *PF-KEY = 'ENTR'
```

```
      IF #PERSONNEL-ID EQ ' '
        MOVE 'LIST' TO #COMMAND
      ELSE
        MOVE 'INFO' TO #COMMAND
      END-IF
    END-IF
    MOVE *PF-KEY TO #PF-KEY
*
  VALUE 'COMMAND' IGNORE              /* Local command handling
  VALUE 'PARM-END' IGNORE             /* End of parameter parsing
  VALUE 'START'  IGNORE
  VALUE 'END'    IGNORE
  NONE  IGNORE
END-DECIDE
END-SUBROUTINE
*
*  General Subroutines
*
DEFINE SUBROUTINE PARM-OPTION
*
* Employee name is an accepted parameter
* either with keyword NAME or as first parameter.
* Employee number is  accepted with keyword NUMBER.
*
DECIDE ON FIRST VALUE OF #PARM-KEYWORD
  VALUE '1','NAME'     MOVE #PARM-VALUE TO #NAME
  VALUE 'NUMBER'
    IF  #PARM-VALUE IS (N8)
      MOVE RIGHT #PARM-VALUE TO #PERSONNEL-ID
    ELSE
      MOVE 1 TO #OUTPUT-ERROR-CODE     /* error
      MOVE 6801 TO #ERROR-NUMBER       /* invalid parameter
      ESCAPE ROUTINE
    END-IF
  NONE IGNORE
END-DECIDE
END-SUBROUTINE
*
DEFINE SUBROUTINE LINE-OPTION
*
* Move the relevant data from the list line into our
* program data
*
MOVE #LINE-PERSONNEL-ID TO #PERSONNEL-ID
MOVE #LINE-NAME         TO #NAME
END-SUBROUTINE
*
*
DEFINE SUBROUTINE ITEM-OPTION
*
* Move the relevant data from the list line into ITEM-NAME
*
```

```
  COMPRESS 'NUMBER = ' #LINE-PERSONNEL-ID INTO #ITEM-NAME
  END-SUBROUTINE
  *
  END
```

Defining a User Command

Every command defined for Open NSPF is implemented by an Open NSPF routine. The Open NSPF routines are of type Natural subprogram with a fixed parameter area to communicate with Natural ISPF:

```
DEFINE DATA
PARAMETER USING ISP-UC-A      /* Standard Open NSPF interface
PARAMETER
1 #STATIC-DATA        (A253)
1 #GLOBAL-DATA        (A32)   /* Shared data for Open NSPF routine
LOCAL ......
END-DEFINE
```

The parameter area `ISP-UC-A` can be found in the User Exit Library (`SYSISPX`). The Open NSPF routine is called from Natural ISPF every time the command is issued.

Site Control Table: Adding a User Command

The Site Control Table can be found in the User Profile Library and is usually called `CONTROLU`. In this table, you can define new commands.

Edit macro `MAC-CNFZ` is available when editing the Site Control Table. If you wish to use this edit macro, you must use the Natural utility `SYSMAIN` to copy the following programs from the Example Library (`SYSISPE`) to the User Profile Library (`SYSISPFU`):

```
MAC-CNF*
MACCNF*
```

Note: As an alternative, it would also be sufficient to define the library `SYSISPE` as a `STEPLIB` for the library `SYSISPFU`.

- If you wish to create a new `CONTROLU` member, you can use the edit macro using the function command

```
EDIT CNF CONTROLU MODEL=MAC-CNFZ
```

- If you wish to modify an existing `CONTROLU` member, use the following command in the edit session with the existing member:

```
REGENERATE
```

To add a new command to Natural ISPF, proceed as follows:

1. Allocate a two-letter code to the command.
2. Prepare a Natural subprogram to handle the command and copy it into `SYSLIB`.
3. Add the user command to the Site Control Table.

Once the command has been entered in the Site Control Table and the corresponding subprogram has been copied to `SYSLIB`, the subprogram is executed every time a user issues the command.

The command attributes are entered into one line in the Site Control Table in fixed positions with the exclamation mark (!) in the beginning of the line.

Example:

```
*COMMAND     !
*            !SECURITY OPTION/LEVEL
*            !  !COMMAND-TYPE
*            !  ! !MIN ABBV
*            !  ! ! !PROGRAM
*            !  ! ! !        !PROGRAM-PARM
*            !  ! ! !        !    !SUBSYSTEM
!MAIL        !  !U!4!ML      !    !
```

Note: The column delimiting character `!` used in the above example is keyboard-language dependent and corresponds to hex code `4F`.

Parameter	Meaning
`Command`	Full command name, for example: `MAIL`.
`Security Option/Level`	One character security option with one digit for level. The command will be active only if the user has been assigned an authorization level greater or equal to the command level (e.g. `Q2`). If left blank, the command is always active. The one-character security option is the Authorization class (see *Authorization Classes* in the *Natural ISPF Administration Guide*). To restrict access to this object/function you should use the '=' (`USER DEFINED`) authorization class and assign different authorization levels to user/user groups. The one-digit level corresponds to the authorization level defined for the specified class in the user authorization table (see the section *User Definitions* in the *Natural ISPF Administration Guide*).

Parameter	Meaning
Command-Type	Identification of a user-defined command. This column *must* contain the letter U for every definition of a user-defined command.
Min abbv	Minimum characters in command line to identify the command. For example, 2 would allow users to enter MA. 4 allows no command abbreviation for MAIL.
Program	Two-character code to be used for the subprogram name. It is strongly recommended that you use a letter for the first digit. For example, a code of ML means the subprogram must be called ISUCML.
Program-Parm	For future use.
Subsystem	One-character subsystem code. The codes are the same as in the Configuration Table. The command will be active if the subsystem is installed. For example, M means the user command is available to z/OS users. If left blank, the command is always active. For a list of available subsystems, see *Subsystems Supported by Natural ISPF* of the *Natural ISPF Administration Guide*.

Parameter Description

Parameter Name	Length	Type
COMMAND	(A128)	Input/Output

This field contains the command in full length which the user typed in to invoke the Open NSPF routine, including those parameters that precede the first parameter delimiter.

Parameter Name	Length	Type
COMMAND-PARM	(A64)	Input/Output

Command parameters which were entered by the user after the first parameter delimiter.

For example, assuming the parameter delimiter is a comma (,), and the user-defined command is UCOM, the COMMAND and COMMAND-PARM fields have the following contents:

Command typed in by user	Value for COMMAND parameter	Value for COMMAND-PARM parameter
UCOM	UCOM	(blank)
UCOM A	UCOM A	(blank)
UCOM A,X	UCOM A	X
UCOM A B,X	UCOM A B	X
UCOM A,X,Y	UCOM A	X,Y

Parameter Name	Length	Type
ERROR-NUMBER	(N4)	Output

An error number which is reported to the user. The error is brought in the field TITLE and is available in the PERFORM and DISPLAY events so that it can be presented to the user. See also the field OUTPUT-ERROR-CODE. If OUTPUT-ERROR-CODE is not set (value is zero), information can be passed to the user since the current function is not aborted. The error text is taken according to number ranges from the following libraries:

```
6800 - 8999: SYSISPS1
9000 - 9999: are reserved for the user in SYSISPS1.
```

Parameter Name	Length	Type
ERROR-TEXT	(A75)	Output

Text to be displayed. If this field is filled, ERROR-NUMBER is ignored.

Parameter Name	Length	Type
ERROR-PARM	(A75)	Output

The ERROR-PARM tokens delimited by a semicolon (;). Parameters to be substituted in the error texts are denoted as :1: :2:

Parameter Name	Length	Type
GLOBAL-DATA	(A32)	Input/Output

Data Area common to all Open NSPF routines.

Parameter Name	Length	Type
STATIC-DATA	(A253)	Input/Output

Shared data segment identified by STATIC-ID. The data is always updated when it is changed upon return to Natural ISPF. The data segment lives as long as the Natural ISPF session lives. If the STATIC-DATA and STATIC-ID are changed in one operation, the data is updated for the old ID and the new segment for the new ID is returned.

Parameter Name	Length	Type
STATIC-ID	(A2)	Input/Output

Identification for a shared data segment. If this ID is changed, the subprogram is invoked again and the appropriate data segment is returned.

Parameter Name	Length	Type
OUTPUT-ERROR-CODE	(N3)	Output

A non-zero value denotes an error situation to Natural ISPF, that is, the current function is aborted and the error denoted by the fields ERROR-NUMBER, ERROR-TEXT and ERROR-PARM is reported in the previous screen. This should be used in real error situations.

Examples

The first example program is relevant to sites that run Software AG's Office System Con-nect. It checks for new items in the user's Con-nect Inbasket.

```
*  Program checks whether something new is in
*  CON-NECT  inbasket
DEFINE DATA
PARAMETER USING ISP-UC-A
PARAMETER
1 #STATIC-DATA(A253)
1 #GLOBAL-DATA(A32)
LOCAL
1 #RC        (N2)
1 #CAB       (A8)
1 #PSW       (A8)
1 #PHONE     (P8)
1 #MAIL      (P8)
1 #INVIT     (P8)
1 #OP-MAIL   (P8)
1 #POST-MAIL (P8)
END-DEFINE
MOVE *USER TO #CAB
CALLNAT 'Z-INBKT' #RC #CAB #PSW #PHONE #MAIL #INVIT #OP-MAIL #POST-MAIL
IF #RC NE 0
  MOVE 'Connect error' TO #ERROR-TEXT
  MOVE 1               TO #OUTPUT-ERROR-CODE
ELSE
  MOVE  'You have' TO #ERROR-TEXT
  DECIDE FOR EVERY CONDITION
    WHEN #PHONE NE 0
      COMPRESS #ERROR-TEXT #PHONE 'phones' INTO #ERROR-TEXT
    WHEN #MAIL NE 0
      COMPRESS #ERROR-TEXT #MAIL 'mail' INTO #ERROR-TEXT
    WHEN #INVIT NE 0
      COMPRESS #ERROR-TEXT #INVIT 'Invitation' INTO #ERROR-TEXT
    WHEN NONE
      COMPRESS #ERROR-TEXT 'No mail' INTO #ERROR-TEXT
  END-DECIDE
END-IF
END
```

The second example program is relevant to BS2000 sites. It translates the command `FS(TAT)` into the Natural ISPF command `LIST BF` to list BS2000 files. In this way, `FS` and `FSTAT` become synonyms of the Natural ISPF command `LIST BF *`.

```
* This program translates command FS(TAT) ... into LIST BF ..
* to list BS2000 files
DEFINE DATA
PARAMETER USING ISP-UC-A
PARAMETER
1 #STATIC-DATA(A253)
1 #GLOBAL-DATA(A32)
LOCAL
1 #WRITTEN-CMD (A128)
1 #FUNC-PARMS  (A128)
1 #-DEL        (A1) CONST <H'FE'>
END-DEFINE
*
EXAMINE #COMMAND FOR FULL ' ' REPLACE FIRST WITH #-DEL
SEPARATE #COMMAND LEFT INTO #WRITTEN-CMD #FUNC-PARMS
  WITH DELIMITER #-DEL
IF #FUNC-PARMS = ' '
  MOVE '*' TO #FUNC-PARMS
END-IF
COMPRESS 'LS BF' #FUNC-PARMS INTO #COMMAND
END
```

4 Application Programming Interface

The programs described in this chapter can be used in Natural applications (or in Open NSPF routines) to access Natural ISPF internal information.

All programs and required Natural ISPF modules are loaded into the Exit Library and into the library SYSTEM. If you wish to use the API programs, it is strongly recommended that you define either SYSTEM or SYSISPE as STEBLIB for your application.

ISP-U000 - Current Session Program

This program is supplied in object form only. It is a Natural subprogram which returns information about the current Natural ISPF session and can be called from user programs outside Natural ISPF.

The following parameters must be defined:

Parameter	Format	Type	Meaning
#OBJECT	(A2)	O	Contains 2-character abbreviation of the object type. For a list of possible values, see the *Table of Exits and Object Abbreviations* in section *User Exits* in the *Natural ISPF Administration Guide*.
#FUNCTION	(A2)	O	Contains 2-character abbreviation of the function currently executed.
#SES-DATA	(A253)	O	Contains all parameters; must be redefined according to object type. A special local data area is delivered in source form for this redefinition.

Example

The following example is a short Natural macro object which users can call using the COPY MACRO command from an edit session. It generates a program header and uses program ISP-U000 to obtain the name and library of the program being edited:

```
§ *
§ * MACRO GENERATES A STANDARD PROGRAM HEADER FOR THE PROGRAM
§ * BEING CURRENTLY EDITED
§ *
§ DEFINE DATA
§ LOCAL USING ISPN---L  /* Redefinition of Natural data
§ LOCAL
§ 1 #OBJECT (A2)
§ 1 #FUNCTION(A2)
§ 1 #DATA (A253)
§ 1 #PROGRAM (A8)
§ 1 #TEXT (A50/5)
§ 1 #I (N2)
§ END-DEFINE
§ *
§ * GET NATURAL SESSION DATA
```

```
§ *
§ CALLNAT 'ISP-U000' #OBJECT #FUNCTION #DATA
§ MOVE #DATA TO #SES-DATA-N          /* Move to redefinition
§ MOVE #MEMBER TO #PROGRAM
§ SET KEY PF3
§ SET CONTROL 'WL70C12B005/005F'
§ INPUT (AD=MIL'_')
§ WITH TEXT '----- PROGRAM HEADING INFORMATION -------------'
§ 'PROGRAM:' #PROGRAM (AD=OI)
§ 'LIBRARY:' #LIBRARY (AD=OI)
§ / 'PURPOSE:' #TEXT (1)
§ / ' ' #TEXT (2)
§ / ' ' #TEXT (3)
§ / ' ' #TEXT (4)
§ / ' ' #TEXT (5)
§ IF #TEXT(1) = ' '
§ REINPUT WITH TEXT 'PURPOSE IS REQUIRED'
§ END-IF
************************************************************************
* PROGRAM: §#PROGRAM DATE CREATED: §*DATD BY: §*USER
* ---------------------------------------------------------------------
* PURPOSE:
§ FOR #I = 1 TO 5
§ IF #TEXT(#I) NE ' '
* §#TEXT(#I)
§ END-IF
§ END-FOR
* ---------------------------------------------------------------------
* PROGRAM HISTORY
* DATE USER-ID REF-NO DESCRIPTION
 ***********************************************************************
*
```

The following figure shows the result of the `INPUT` statement in the macro when the program is invoked from an edit session with program `NEWPROG` in library `NSPFWORK`:

```
EDIT-NAT:NSPFWORK(NEWPROG)-Program->Struct-Free-29K - >>> Versioning is invoked
 COMMAND===> COPY MAC MAC-HEAD                                  SCROLL===> CSR
****** ***************************** top of data ****************************
****** **************************** bottom of data ***************************

     +-------------------------------------------------------------------+
     ! ----- PROGRAM HEADING INFORMATION ------------                     !
     ! PROGRAM: NEWPROG  LIBRARY: NSPFWORK                               !
     ! PURPOSE: This program demonstrates something _____________        !
     !          ____________________________________________________     !
     !          ____________________________________________________     !
     !          ____________________________________________________     !
     !          ____________________________________________________     !
     !                                                                   !
     ! Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10- !
     !       Help  Split End   Suspe Rfind Rchan Up    Down  Swap  Left  !
     +-------------------------------------------------------------------+




Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Split End   Suspe Rfind Rchan Up    Down  Swap  Left  Right Curso
```

Another useful example called `COPYSYS` can be found in the Example Library. This program copies a member which is edited to the library `SYSLIB`.

ISP-U001 - Access Shortlibs Program

This program allows user programs to access short library names defined in Natural ISPF either in the user profile or in the global shortlib table. If an entry is found, the data set name and volser of the data set is returned.

Parameter	Format	Type	Meaning
#IN-SHORT	(A2)	I	2-byte library abbreviation.
#OUT-DSN	(A44)	O	Data set name, if given abbreviation was found.
#OUT-VOLSER	(A6)	O	Volser, if defined in profile.

ISP-U002 - Retrieve Error Texts Program

This program can be used to retrieve error texts from the Natural ISPF Error Message Library SYSISPS*x* (where *x* is the current language code). If a text is not found for the current language code, the English text is returned. Parameters can be replaced in the error text.

This program can only be used by Open NSPF programs, which are executed from SYSLIB. This program will not work if invoked from any user application.

Parameter	Format	Type	Meaning
#ERR-NUMBER	(N4)	I	Error number to be retrieved.
#ERR-TEXT	(A75)	O	Text, including substituted parameters.
#ERR-PARM	(A75)	I	Parameters separated by (;). Formal parameters in the text must be specified as :*n*:, where *n* is the number of the parameter.

Note: Error messages 9000 - 9999 are reserved for use by your site.

ISP-U003 - Read Data from Edit Session Program

This program can be used to read data from an edit session. When using this program, you must use the ISP-U03A data area which contains the following fields:

Parameter	Format	Type	Meaning
#LINES	(A132/20)	O	Data lines from the Editor session, a maximum of 20 lines can be read in one call.
#ED-SESNUM	(B2)	I	Editor session number.
#ERROR-NUMBER	(N4)	O	Error number, 0 if function was executed correctly.
#LINES-READ	(N2)	O	Number of lines returned in field #LINES.
#LINES-TO-READ	(N2)	I/O	Number of lines to read in 1 call
#LINE-LENGTH	(N4)	O	Length of the lines.
#ED-NUM	(N6)	W	Do not modify between calls.
#EOF	(L)	O	True if the last lines have been read.
#TRACE	(L)	O	If true, the number of lines, the information about the last line and the first 60 characters of each line read are displayed.

Example:

```
DEFINE DATA
LOCAL USING ISP-U03A
LOCAL
1 #I    (N2)
END-DEFINE
INPUT #ED-SESNUM
*
MOVE 20 TO   #LINES-TO-READ
*
WRITE 'user processing of a macro result' //
*
REPEAT
  CALLNAT 'ISP-U003' #READ-PARM
  IF #ERROR-NUMBER NE 0
    WRITE '=' #ERROR-NUMBER
    ESCAPE BOTTOM
  END-IF
  FOR #I = 1 TO #LINES-READ
    PERFORM LINE-PROCESSING
  END-FOR
  IF #EOF
    ESCAPE BOTTOM
  END-IF
END-REPEAT
*
DEFINE SUBROUTINE LINE-PROCESSING
PRINT  #LINES(#I)
END-SUBROUTINE
END
```

ISP-U004 - Pass Command Script Program

This subprogram can be used to pass a command script (that is, a sequence of Natural ISPF commands) to Natural ISPF and to execute it when Natural ISPF is invoked next time. See also member `ISP-COAP` in the example library.

Parameter	Format	Type	Meaning
#IN-COMMANDS	(A50/10)	I	Command lines to be processed.
#IN-LINES	(N2)	I	Number of lines to be processed in IN-COMMANDS.
#COMMAND-POS	(A1)		For future use.
#FUNCTION	(A1)		For future use.
#RC		O	Return code: 0 - Processing terminated ok.

Parameter	Format	Type	Meaning
			1 - Natural ISPF is not active. 2 - Processing terminated with error.

ISP-U005 - Check for Natural Member Versions Program

Subprogram checks whether versions (update decks) for a Natural member exist.

Parameter	Format	Type	Meaning
#IN-LIBRARY	(A8)	I	Natural library name.
#IN-MEMBER	(A8)	I	Natural member name.
#RC	(N4)	O	Return code: 0 - Processing terminated ok.
#OUT-VERSION	(L)	O	TRUE Versions for this member exist.
#GEN-VERSION	(L)	O	FALSE Natural ISPF versioning is not active in this environment.

ISP-U006 - Set Source Area Attributes Program

Subprogram sets source area attributes and is used in macro processing.

Parameter	Format	Type	Meaning
#IN-MEMBER	(A8)	I	Member name to be set.
#IN-TYPE	(A1)	I	Natural object type to be set.
#IN-MODE	(A1)	I	Programming mode (S/R) to be set.

ISP-U007 - Check User Authorization Program

This subprogram checks whether the current user is authorized to use Natural ISPF or certain parts of its functionality.

Parameter	Format	Type	Meaning
#IN-OBJECT	(A2)	I	2-character code of Natural ISPF object being checked (leave blank to check if user is authorized to access any object).
#IN-FUNCTION	(A2)	I	2-character code of Natural ISPF object being checked (leave blank to check if user is authorized to use any function valid for a specific object; irrelevant if IN-OBJECT is blank).
#RC	(N4)	O	Return code: 0 - User is authorized. -1 - User is not authorized. >0 - "Error in execution". Use USR0320N to retrieve error text from library SYSISP1. = -*NNNN* - Natural runtime error *NNNN* has occurred during execution. Use USR0120N to retrieve error text.

ISP-U008 - Current Session Program Including Global Data

This program is supplied in object form only. It is a Natural subprogram which returns information about the current Natural ISPF session and can be called from user programs outside Natural ISPF. It works like ISP-U000 but includes global data

The following parameters must be defined:

Parameter	Format	Type	Meaning
#OBJECT	(A2)	O	Contains 2-character abbreviation of the object type. For a list of possible values, see the *Table of Exits and Object Abbreviations* in the section *User Exits* in the *Natural ISPF Administration Guide*.
#FUNCTION	(A2)	O	Contains 2-character abbreviation of the function currently being executed.
#CURR-SES-NUM	(N2)	O	Contains session number of SAG edit session.
#L-COMMAND	(A50)	O	Contains command line to be interpreted.
#MACRO-CHAR	(A1)	O	Contains character which will be interpreted as macro character.
#CMD-DEL	(A1)	O	Contains command delimiter character for command line.
#SES-DATA	(A253)	O	Contains all parameters; must be redefined according to object type. A special local data area is delivered in source form for this redefinition

ISP-U009 – Current Session Program / Previous Session Program

This program is supplied in object form only. It is a Natural subprogram which returns information about the current Natural ISPF session and can be called from user programs outside Natural ISPF. It works in the same way as `ISP-U000` except when executing a Natural program. In this case, the data of the previous session is returned.

The following parameters must be defined:

Parameter	Format	Type	Meaning
#OBJECT	(A2)	O	Contains 2-character abbreviation of the object type. For a list of possible values, see the *Table of Exits and Object Abbreviations* in section *User Exits* in the *Natural ISPF Administration Guide*.
#FUNCTION	(A2)	O	Contains 2-character abbreviation of the function currently executed.
#SES-DATA	(A253)	O	Contains all parameters; must be redefined according to object type. A special local data area is delivered in source form for this redefinition.

5 Authorization

This chapter lists the available authorization classes and the Natural ISPF objects they refer to.

- Authorization classes as they appear in the Class column of the table are assigned authorization levels in user definitions (see the *Natural ISPF Administration Guide*).
- The codes in the Code column of the table are used in menu lines in menu definitions, as well as in the Site Control Table.

Code	Class	Natural ISPF Objects
1	z/VSE/ESA files	z/VSE/ESA files
2	z/VSE/ESA SYSOUT	SYSOUT of z/VSE/ESA jobs
3	Active jobs	Active jobs (z/OS and z/VSE/ESA)
4	BS2000 jobs	BS2000 jobs and job variables
9	Operator commands	Use of operator commands
A	NSPF Administrator	Configuration operations
B	BS2000 files	BS2000 files
D	Data set maintenance	Data sets and volumes (z/OS)
E	BS2000 LMS elements	LMS library elements and previous versions
J	SYSOUTS	Jobs, including SYSOUT files (z/OS)
L	CA Librarian	CA Librarian members and versions
N	Natural programming	Natural objects and views
P	PDS maintenance	PDS members and previous versions
S	System info	System operations
T	CA Panvalet	CA Panvalet members and previous versions
=	User defined	Site-specific menu options